



Intel® Edison

Native Application Guide

February 2015

Revision 003



Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication, or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

All Code placed under the MIT License. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE

The main algorithm has been derived from <https://github.com/bagilevi/android-pedometer>, which has copyleft license (without any real license). Still this item needs to be checked with legal to make sure there are no compatibility issues. Some functions have been taken from MPU6050 specific libraries which shares the same licensing conditions with this code.

Some of the code is taken or derived from i2c-dev.h - i2c-bus driver, char device interface (Copyright ©1995-97 Simon G. Vogl Copyright © 1998-99 Frodo Looijaard frodol@dds.nl) and from i2c.h (Copyright © 2013 Parav Nagarsheth), which is under GNU General Public License as published by the Free Software Foundation.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

The code names presented in this document are only for use by Intel to identify products, technologies, or services in development that have not been made commercially available to the public, i.e., announced, launched, or shipped. They are not "commercial" names for products or services and are not intended to function as trademarks.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com/design/literature.htm>.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

Intel and the Intel logo are trademarks of Intel Corporation in the US and other countries.

* Other brands and names may be claimed as the property of others.

Copyright © 2015 Intel Corporation. All rights reserved.



Contents

1	Introduction	6
1.1	References	6
1.2	Acronyms and abbreviations	7
2	Native Application Development	8
2.1	Setting up the host machine	8
2.2	Install the Intel® Edison toolchain	8
2.3	Configure host environment	9
2.4	Develop a simple application	9
2.5	Deploy application to target device	10
2.5.1	Deploy application binary with SCP	10
2.5.2	Deploy application binary with SFTP	10
2.6	Debugging	10
2.6.1	Onboard debugging	10
2.6.2	Remote debugging	11
3	Using Eclipse	12
3.1	Eclipse configuration	12
3.2	Set up the Yocto plugin	12
3.3	Set up toolchain location in Eclipse	14
3.4	Set up remote hardware	15
3.5	Creating a new project on Eclipse	19
3.6	Development process on Eclipse	21
3.7	Deployment with Eclipse	22
3.8	Debugging with Eclipse	24
4	Porting an existing project to Intel® Edison	26
4.1	Using external libraries	27
5	Simple Native Applications	28
5.1	Windows native applications	28
5.2	Linux native applications	28
5.3	Sample GPIO Write Application	29
6	Sample Pedometer Application	31
6.1	Reading accelerometer raw data	32
6.1.1	I ² C operations	32
6.1.2	Communication with MPU6050	34
6.1.3	Reading raw data	36
6.1.4	Pedometer algorithm	36
6.2	Saving data and distance, calorie calculation	38



Figures

Figure 1	Toolchain download files	8
Figure 2	Add repository	13
Figure 3	Select Yocto plugin.....	13
Figure 4	Preferences window.....	14
Figure 5	Open perspective	15
Figure 6	New connection	15
Figure 7	New connection details	16
Figure 8	Active provider	16
Figure 9	Transfer with ssh	17
Figure 10	Enter password.....	17
Figure 11	A connected Edison file system.....	18
Figure 12	Select a new project.....	19
Figure 13	A new C project	19
Figure 14	C project basic settings.....	20
Figure 15	Defined toolchain for the new project.....	20
Figure 16	Project folder structure	21
Figure 17	Configure project.....	22
Figure 18	Run configurations.....	23
Figure 19	Problem occurred	23
Figure 20	Run the binary from the Intel® Edison device's shell.....	24
Figure 21	Debug configurations	24
Figure 22	Debug.....	25
Figure 23	Confirm perspective switch	25
Figure 24	Hardware required for the pedometer application	31
Figure 25	Software block diagram	31

Tables

Table 1.	Terminology.....	7
----------	------------------	---



Revision History

Revision	Description	Date
ww33	Initial release.	August 14, 2014
001	First public release.	September 9, 2014
002	Fixed incorrect link.	October 6, 2014
003	Validated code and made minor corrections.	February 4, 2015

§



1 Introduction

This document is written for software developers who are developing native software applications with C and C++ on the Edison Development platform. It covers basic preparation to set up your host to develop apps for the Intel® Edison Development Board, and provides a sample application for a pedometer.

The reader should have a basic understanding of C and C++ software development and knowledge of the Linux operating system.

1.1 References

Reference	Name	Number/location
331188	Intel® Edison Board Support Package User Guide	
331189	Intel® Edison Compute Module Hardware Guide	
331190	Intel® Edison Breakout Board Hardware Guide	
331191	Intel® Edison Kit for Arduino* Hardware Guide	
331192	Intel® Edison Native Application Guide	(This document)
329686	Intel® Galileo and Intel® Edison Release Notes	
331438	Intel® Edison Wi-Fi Guide	
331704	Intel® Edison Bluetooth* Guide	
[YPQSG]	Yocto Project Quick Start Guide	http://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html
[YDM]	Yocto Developer Manual	http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html
[YKDM]	Yocto Kernel Developer Manual	http://www.yoctoproject.org/docs/latest/kernel-dev/kernel-dev.html
	Yocto Project	http://www.yoctoproject.org/docs/1.5.1/dev-manual/dev-manual.html
	GPIO Linux documentation	https://www.kernel.org/doc/Documentation/gpio/sysfs.txt
	GNU Automake	http://www.gnu.org/software/automake/
	GCC Cross Compiler wiki	http://wiki.osdev.org/GCC_Cross-Compiler
	GDB (GNU debugger)	https://www.gnu.org/software/gdb/
	MPU6050 Product Specification:	http://www.invensense.com/mems/gyro/documents/PS-MPU-6000A-00v3.4.pdf
	MPU6050 resources	http://playground.arduino.cc/Main/MPU-6050
	I2C Linux documentation	https://www.kernel.org/doc/Documentation/i2c/dev-interface
	I2C SMBus-protocol documentation	https://www.kernel.org/doc/Documentation/i2c/smbus-protocol
	MPU6050 library	https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050



1.2 Acronyms and abbreviations

Table 1. Terminology

Term	Definition
FTP	File Transfer Protocol
IDE	Integrated development environment
GDB	GNU Debugger
GPL	GNU General Public License
SCP	Secure copy
SFTP	Secure File Transfer Protocol
ssh	Secure shell

§



2 Native Application Development

The processes in this section have been verified on an Ubuntu* 12.04 host machine.

2.1 Setting up the host machine

Install the required packages before you install the Intel® Edison development board toolchain on your Ubuntu host PC. Required packages for C/C++ development on an Ubuntu host machine include:

- gcc
- g++
- gcc-multilib
- build-essential

If you will work with makefiles, you must also install the following required GNU tools:

- make
- automake

Install the required packages with the following command:

```
sudo apt-get install make automake gcc g++ build-essential gcc-multilib
```

2.2 Install the Intel® Edison toolchain

Go to <https://communities.intel.com/docs/DOC-23242> and download the appropriate Intel® Edison Development Platform toolchain installer file for your platform.

Figure 1 Toolchain download files

Link	Software	Operating System	Board	File Size	File Type	Notes
Download	Edison Yocto complete image	Linux - Yocto	Edison	105MB	zip	
Download	Edison SDK Linux 32	Linux	Edison	380MB	zip	
Download	Edison SDK Linux 64	Linux	Edison	382MB	zip	
Download	Edison SDK Mac OSX	OSX	Edison	347MB	zip	
Download	Edison SDK Windows 32	Windows 32	Edison	421MB	zip	
Download	Edison SDK Windows 64	Windows 64	Edison	425MB	zip	
Download	Edison Linux source files	Linux	Edison	34MB	tgz	
Download	Edison GPL/LGPL source files	Linux	Edison	603MB	tar	

The installer zip file will be *edison-sdk-**<host_arch>**-weekly-XX.zip* (where **<host_arch>** is the host computer's operating system and XX is the week number of the release. Unzip the zip file with the following command:

```
unzip edison-sdk-<host_arch>-weekly-XX.zip
```

The command above extracts the file *poky-edison-eglibc-**<host_arch>**-edison-image-core2-32-toolchain-1.6.sh* into your working directory. You must execute this .sh file to install the toolchain, in the desired path, on your host machine.

```
sudo ./poky-edison-eglibc-<host_arch>-edison-image-core-2-32-toolchain-1.6.sh
Enter target directory for SDK (default: /opt/poky-edison/1.6):
You are about to install the SDK to "/opt/poky-edison/1.6". Proceed[Y/n]?Y
[sudo] password for ubuntu:
Extracting SDK...done
Setting it up...done
SDK has been successfully set up and is ready to be used.
```




2.3 Configure host environment

For convenience, you can initialize the Intel® Edison development board cross-compiler and library paths for cross-compiling your application. The toolchain provides an environment setup file. Execute it from your working terminal with the following command:

```
source /opt/poky-edison/1.6/environment-setup-core2-32-poky-linux
```

Check the settings of the `$CC`, `$CPP`, `$CXX`, and `$LD` variables with the following:

```
echo $CC
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky-edison/1.6/sysroots/core2-32-
poky-linux
echo $CXX
i586-poky-linux-g++ -m32 -march=i586 --sysroot=/opt/poky-edison/1.6/sysroots/core2-32-
poky-linux
echo $CPP
i586-poky-linux-gcc -E -m32 -march=i586 --sysroot=/opt/poky-edison/1.6/sysroots/core2-
32-poky-linux
echo $LD
i586-poky-linux-ld --sysroot=/opt/poky-edison/1.6/sysroots/core2-32-poky-linux
echo $GDB
i586-poky-linux-gdb
```

Note: Using the source command and setting environment variables will only work on the current terminal session where environment-setup-i586-poky-linux was run. Each time you open a new terminal for development purposes, you must set environment variables for the Intel® Edison development board.

2.4 Develop a simple application

A typical first step is to develop a classic *Hello World* sample. Create a *.c file with a text editor (for example, vi hello_world.c) and write the first *Hello World* sample, as shown below:

```
#include <stdio.h>
int main(){
    printf("Hello World\n");
    return 0;
}
```

Compile and build hello_world.c source code with the following:

```
$CC -o hello hello_world.c
```

Similar to the C application, create a *hello_world.cpp* file and implement your hello world cpp code as shown below:

```
#include <iostream>
int main(){
    std::cout<<"Hello World"<<std::endl;
    return 0;
}
```

Compile and build the *hello_world.cpp* source code with the following:

```
$CXX -o hello_cpp hello_world.cpp
```

Note: See chapter 5 for a GPIO write application and chapter 6 for a more sophisticated pedometer/calorie-burning application.



2.5 Deploy application to target device

After you have built the application, you may use any of the following methods to deploy binary to the target device, if the board has connected to the LAN.

2.5.1 Deploy application binary with SCP

You can use a remote file copy program such as SCP (secure copy) that copies files between hosts on a network. SCP uses `ssh` for data transfer; it uses the same authentication, and provides the same security as `ssh`. To transfer files with SCP, do the following:

```
scp file_name username@host.address:/remote/path/to/transfer
```

For the hello world C and CPP samples:

```
scp hello root@192.168.2.15:/home/root/sample
scp hello_cpp root@192.168.2.15:/home/root/sample
```

2.5.2 Deploy application binary with SFTP

In addition to SCP, you may also use SFTP (Secure File Transfer Protocol) to transfer or deploy a binary file to a remote target board. Similar to FTP, SFTP is an interactive file transfer protocol that performs all operations over an encrypted `ssh` transport. To transfer files with SFTP, do the following:

```
sftp username@host.address
sftp> put filename
```

For the hello world C and CPP samples:

```
sftp root@192.168.2.15
sftp> put /path/to/hello
sftp> put /path/to/hello_cpp
```

2.6 Debugging

When developers compile, build, and run their applications, they may encounter runtime errors or other problems. Debugging the generated binary is the best way to troubleshoot these problems. As the preferred debugger on Linux systems, **GNU Debugger** (GDB) is included with the Intel® Edison development board software package. GDB is free software protected by the GNU General Public License.

Note: For more information on GDB, visit: <https://sourceware.org/gdb/current/onlinedocs/gdb.html>.

Use GDB for onboard and remote debugging. Add the `-g` flag to include appropriate debug information on the generated binary to debug the native application with GDB and deploy to the target device. For example:

```
$CC -g -o helloed hello_world.c
$CXX -g -o hello_cppd hello_world.cpp
```

2.6.1 Onboard debugging

Start debugging and run the application with the following:

```
gdb debugthisprogram
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
```



```
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/root/debugthisprogram...done.
(gdb) run
```

It is also possible to set breakpoints, watch variables, and change preprocessors, arguments, and other debugging capabilities. Run the *help* command to view some of the options.

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program.....
```

2.6.2 Remote debugging

When developing native applications, you will use a host machine to cross-compile the application and deploy it on the remote Intel® Edison board. To enable remote debugging, start *gdbserver* (included in the standard Linux* distribution) to open a port on the Intel® Edison board for the host machine, so that developers can debug binaries from the host machine remotely.

On the Intel® Edison device, start *gdbserver* on port 1234:

```
gdbserver :1234 debugthisprogram
Process debugthis created; pid = 2625
Listening on port 1234
```

On the host machine, run *gdb*, then connect target device using *<target_ip>:<port>* and run with *continue* as shown below:

```
gdb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb) target remote 192.168.2.15:1234
Remote debugging using 192.168.2.15:1234
0xb7fd0000 in ?? ()
(gdb) continue
```

3 Using Eclipse

The Eclipse* integrated development environment (IDE) is a good choice as a cross-compile IDE. There is a Yocto plugin for Eclipse that enables native application developers to develop, build, debug, and deploy native applications for the Edison development platform.

Before installing Eclipse, make sure Oracle* Java* has been installed on your Ubuntu host machine. See the Ubuntu community page for detailed instructions (<https://help.ubuntu.com/community/Java>).

Note: We use the standard package of Eclipse Kepler (<https://www.eclipse.org/downloads>) in these examples. Depending upon your host machine configuration, you can download the 32-bit or 64-bit version. Windows users can directly download the Eclipse software from <https://software.intel.com/en-us/iot/downloads>. The whole package comes with a default Eclipse configuration, cross-compiler, and a few sample examples that you can use as a reference.

3.1 Eclipse configuration

You will need to configure Eclipse for C/C++ cross-compile development environments. The Yocto Project Developer Manual provides step-by-step instructions to configure Eclipse for the native application development. For details, see: <http://www.yoctoproject.org/docs/1.5.1/dev-manual/dev-manual.html#adt-eclipse>

3.2 Set up the Yocto plugin

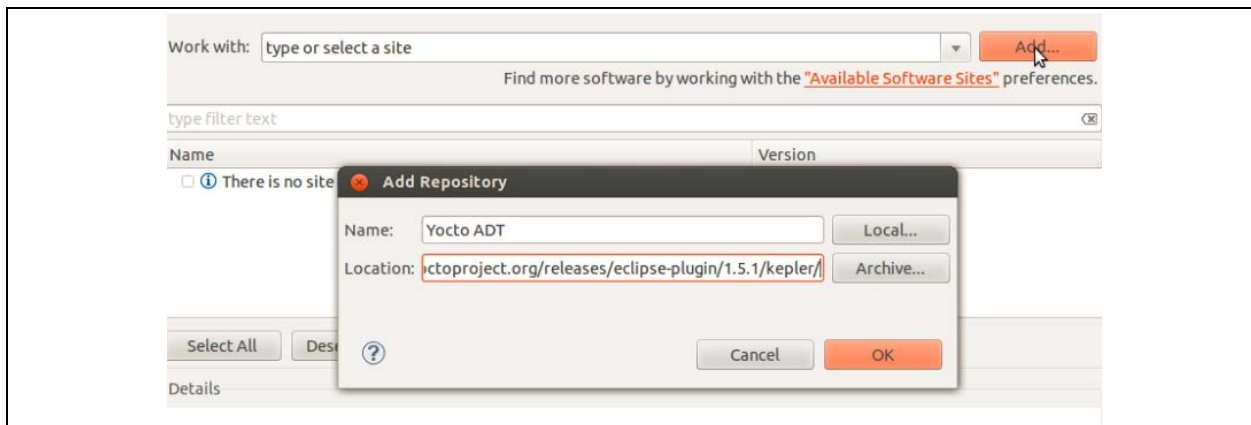
To set up the Yocto plugin, do the following:

1. Start Eclipse and select *Help > Install New Software*. A new window displays.
2. From the *Work with* pull-down menu, select: *Kepler* - <http://download.eclipse.org/releases/kepler>.
3. In the boxes listed below, select the following:
 - a. **Linux tools:**
 - LTTng - Linux Tracing Toolkit
 - b. **Mobile device development:**
 - C/C++ remote launch
 - Remote system explorer end-user runtime
 - Remote system explorer user actions
 - Target management terminal
 - TCF remote system explorer add-in
 - TCF target explorer
 - c. **Programming languages:**
 - Autotools support for CDT and C/C++ development tools
4. Complete the installation of selected plugins and reboot Eclipse.

To install the Yocto plugin for Eclipse, do the following:

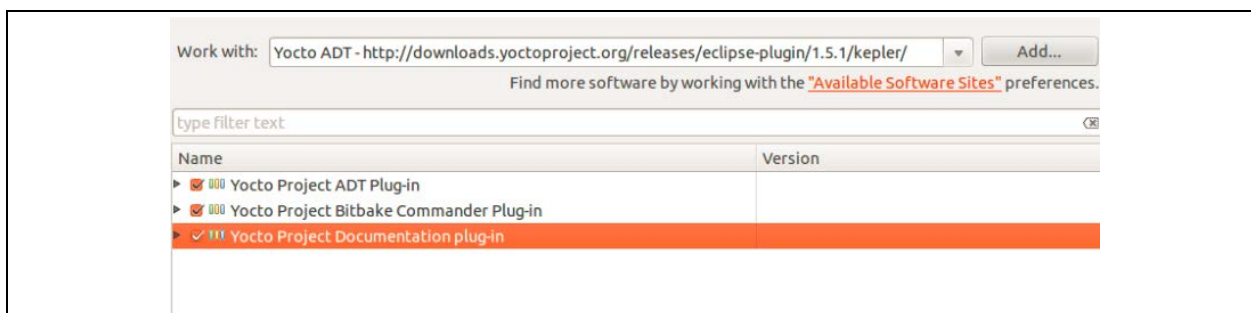
1. Select *Help > Install New Software*. A new window displays.
2. Click the *Add* button to the right of the *Work with* field (Figure 2).
3. In the *Add repository* window, enter a meaningful name like *Yocto ADT* and enter the following link for Eclipse Kepler: <http://downloads.yoctoproject.org/releases/eclipse-plugin/1.5.1/kepler/>.

Figure 2 Add repository



4. Click *OK*.
5. Select the Yocto plugin (Figure 3), then complete the Eclipse plugin installation.

Figure 3 Select Yocto plugin

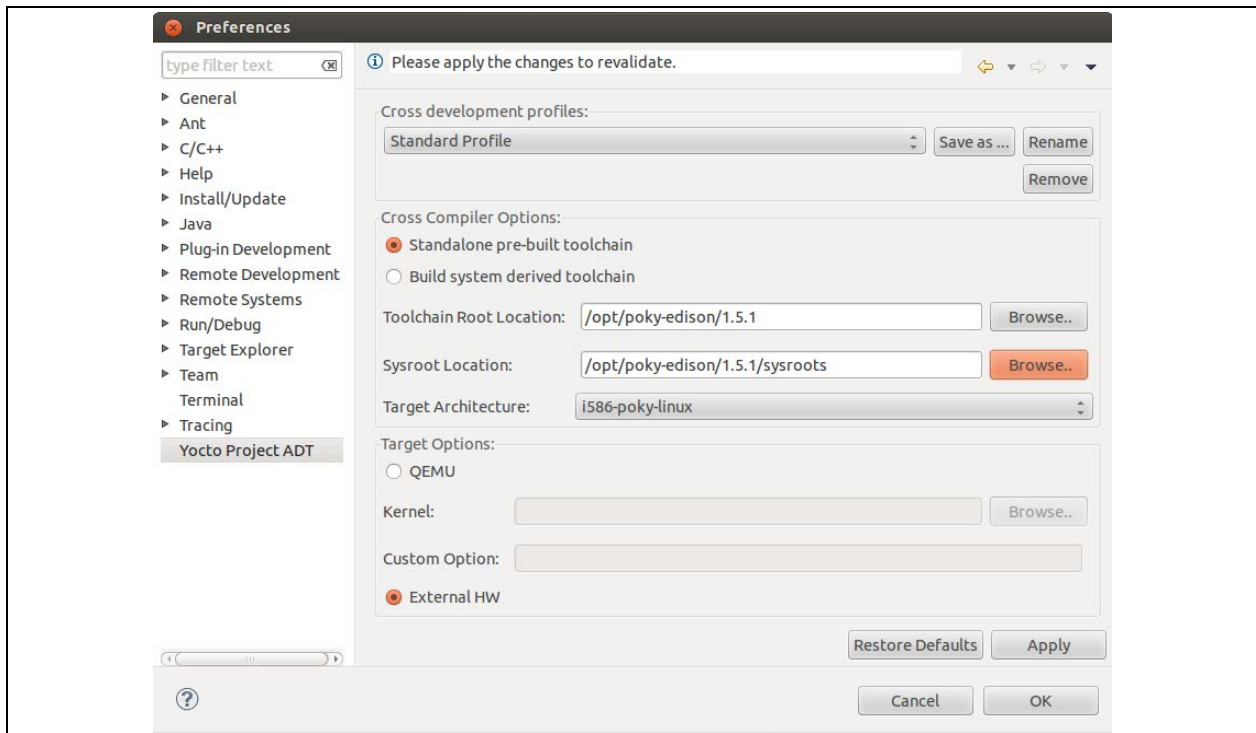


3.3 Set up toolchain location in Eclipse

To set up toolchain locations in Eclipse, do the following:

6. Select *Window > Preferences*. The *Preferences* window displays (Figure 4).
7. In the left column, select *Yocto Project ADT* and enter the Intel® Edison board toolchain location you previously installed.
 - `/opt/poky-edison/1.5.1`
 - `/opt/poky-edison/1.5.1/sysroots`
8. Click *OK*.

Figure 4 Preferences window

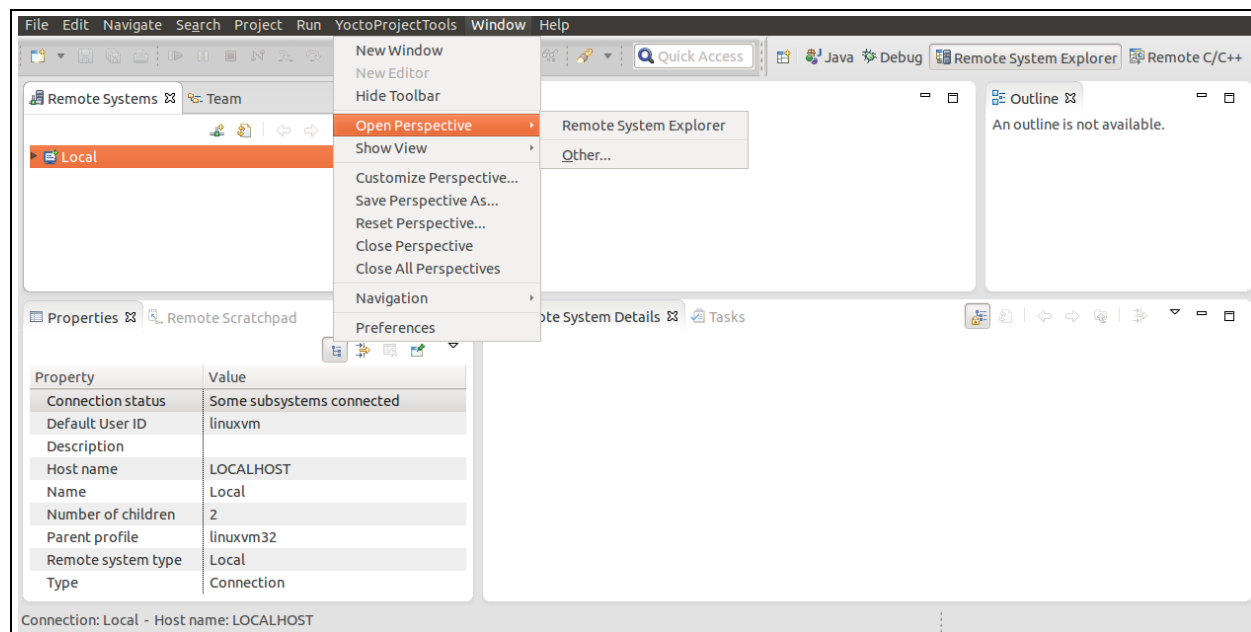


3.4 Set up remote hardware

To make Eclipse automatically deploy the application binary to the Intel® Edison board, you must configure remote hardware. This option simplifies remote debugging.

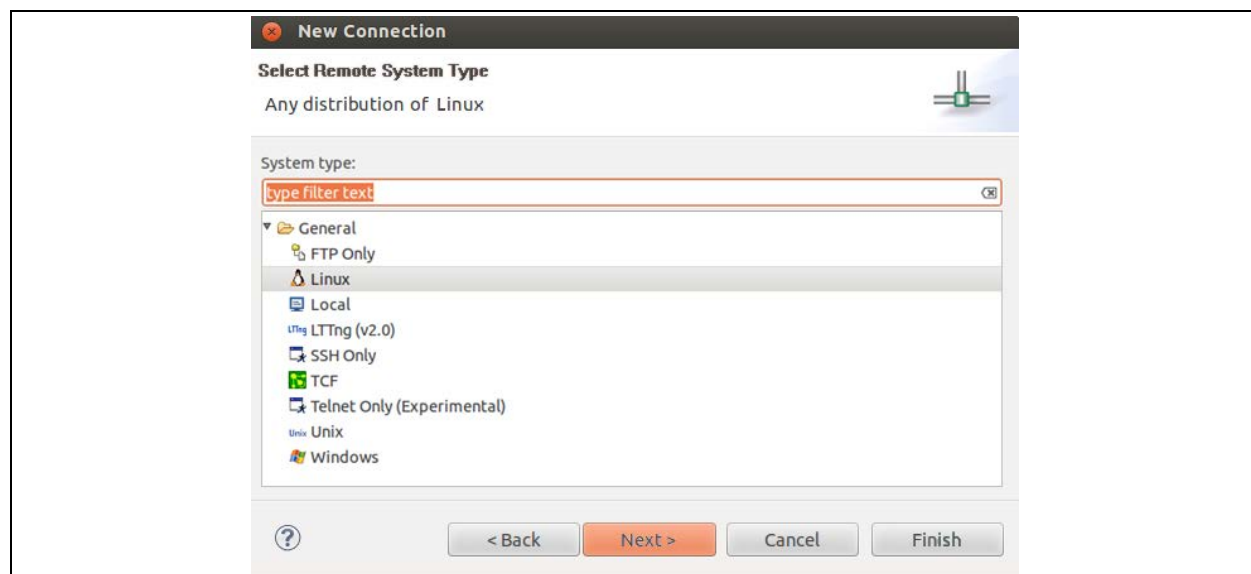
To add the remote target, open the *Remote System Explorer* perspective in Eclipse and select *Window > Open Perspective > Remote System Explorer*.

Figure 5 Open perspective



In the initial state, you only see the local host as a defined system. Define a new connection by clicking on the new connection button on the *Remote Systems* tab. The *New Connection* window displays (Figure 6), allowing you to add the new remote target. (Alternatively, you can select *File > New > Remote* to create a new remote connection.)

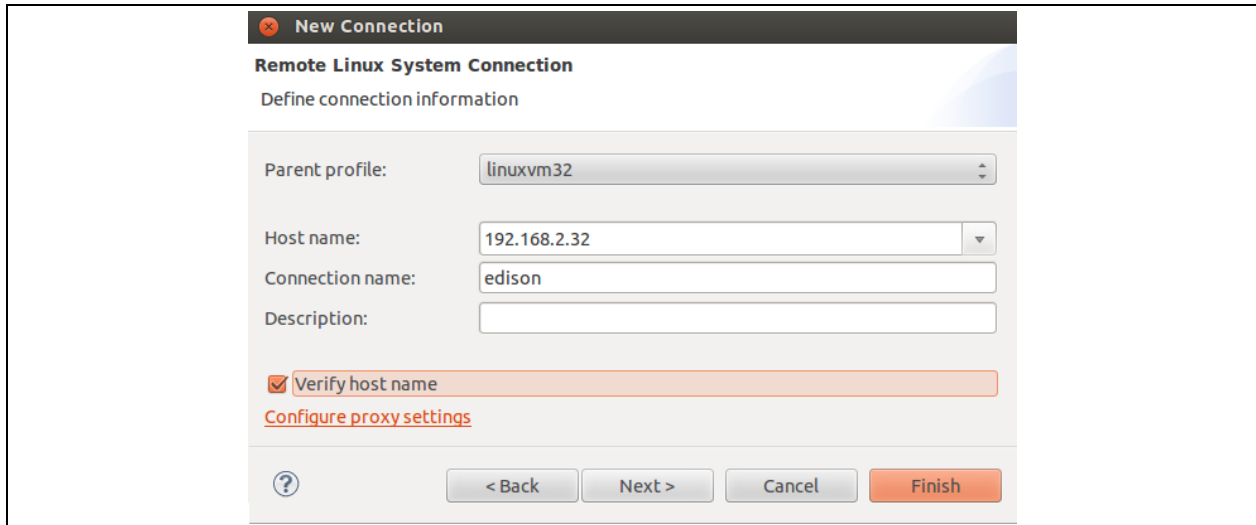
Figure 6 New connection



In the *System type* list, enter a type field descriptor. (In this case, choose *Linux* as shown in Figure 6).

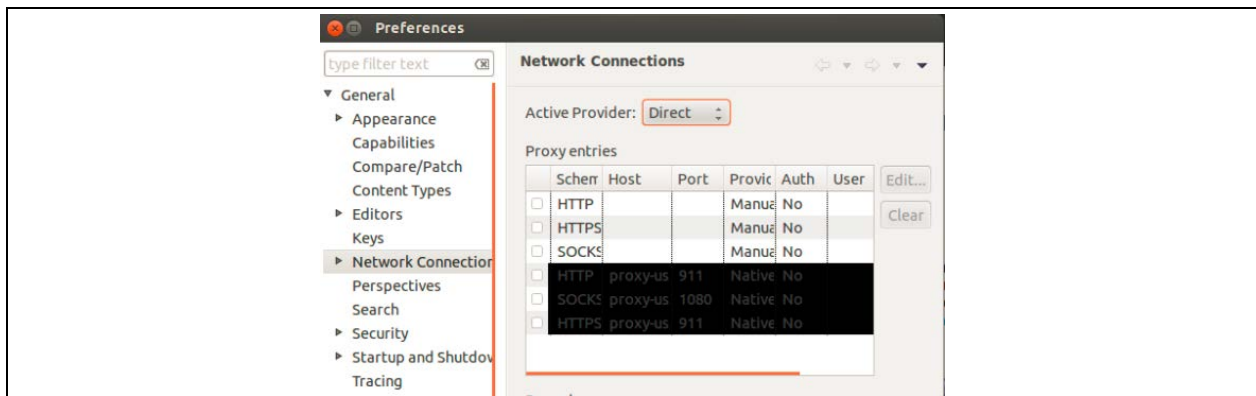
Click *Next* and enter the Intel® Edison device's IP address and connection name (Figure 7).

Figure 7 New connection details



If the host system is using a proxy and Edison is connected via RNDIS USB to connect (using IP address 192.168.2.15), click *Configure proxy settings*. The *Preferences* window displays (Figure 8).

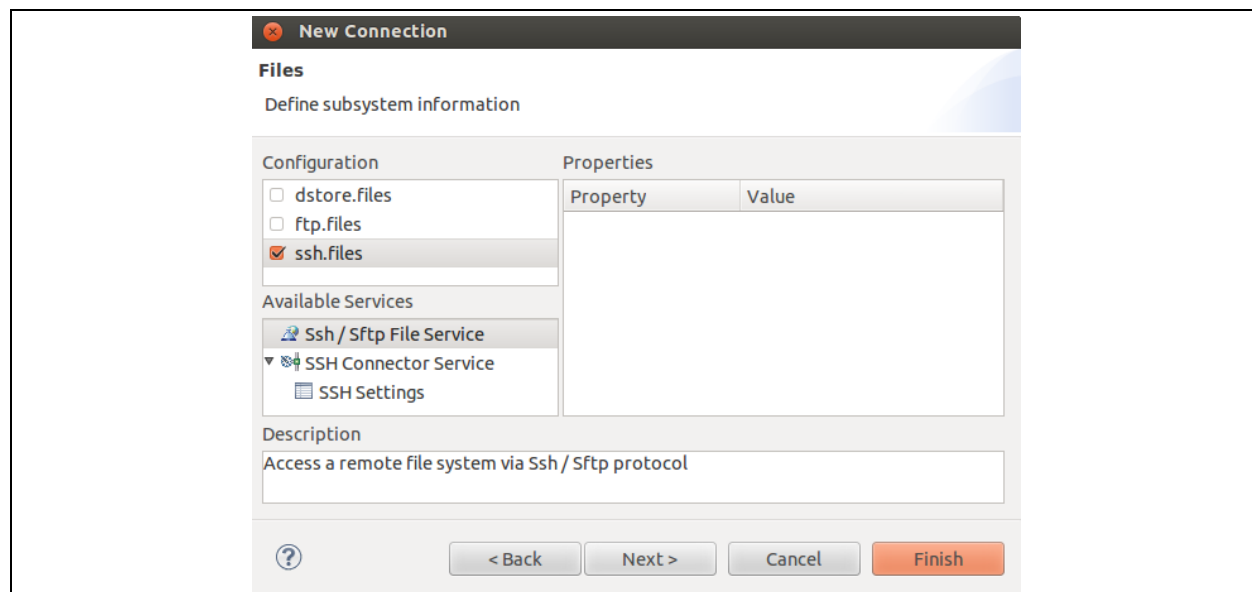
Figure 8 Active provider



Scheme	Host	Port	Provider	Auth	User	Edit...
<input type="checkbox"/> HTTP			Manual	No		
<input type="checkbox"/> HTTPS			Manual	No		
<input type="checkbox"/> SOCKS			Manual	No		
<input type="checkbox"/> HTTP	proxy-us	911	Native	No		
<input type="checkbox"/> SOCKS	proxy-us	1080	Native	No		
<input type="checkbox"/> HTTPS	proxy-us	911	Native	No		

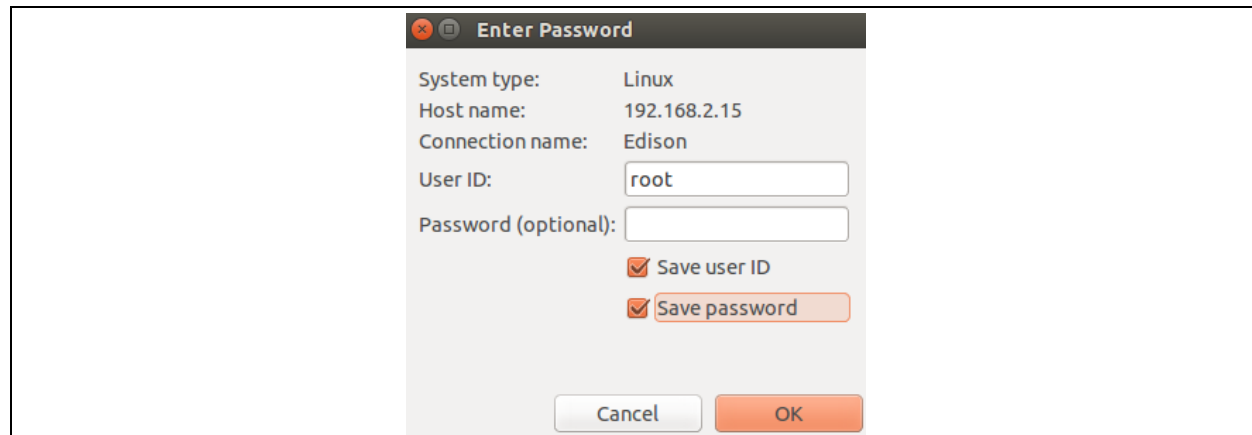
Select *Direct* for *Active Provider*. If your Intel® Edison device is on the network, you can continue to use the native proxy for Eclipse.

Click *Next* and check the *ssh.files* checkbox to transfer/deploy files with the *ssh* protocol (Figure 9).

Figure 9 **Transfer with ssh**


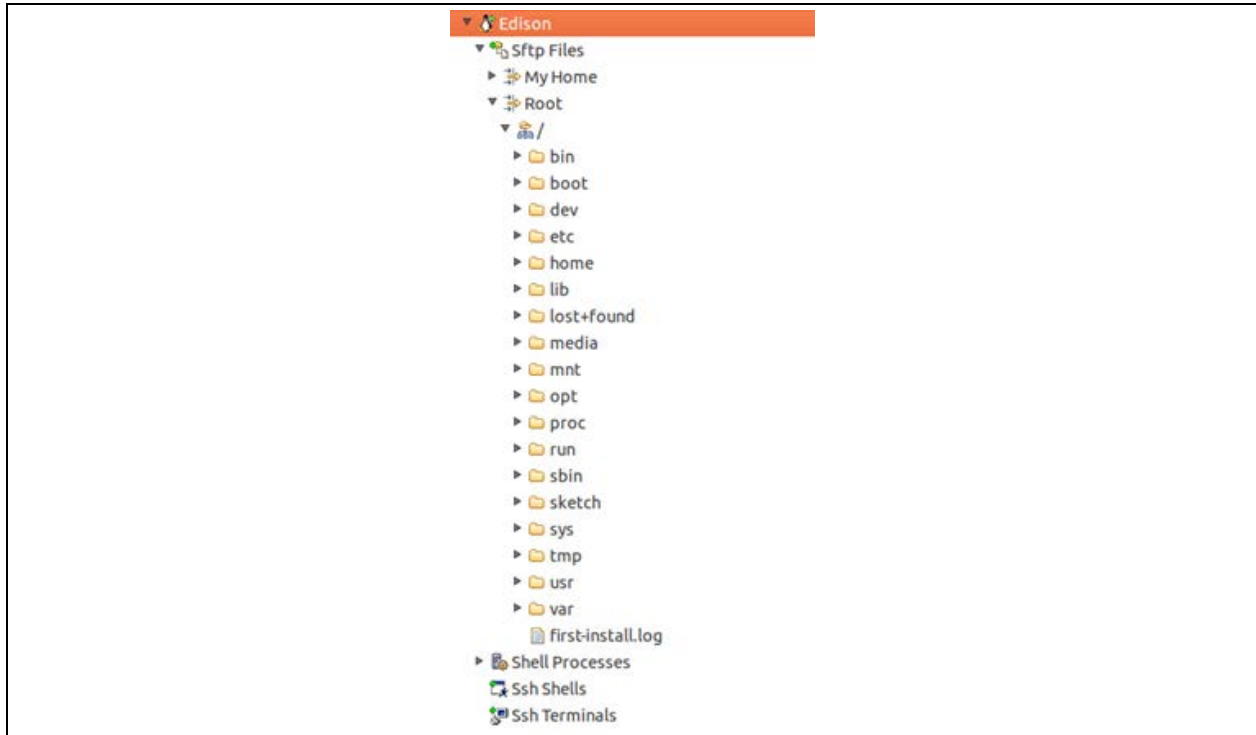
Click *Finish* to complete the new device definition.

The new connection will appear on the connection explorer as *Edison*. Right-click on the device and select *Connect*. A new window displays asking for user ID and password. The new device inherits the user ID from the host. Change the user ID to *root* as the default user and leave the password empty (Figure 10).

Figure 10 **Enter password**


The host will connect to Edison, and developers will be able to see the file system and processes on the device (Figure 11).

Figure 11 A connected Edison file system

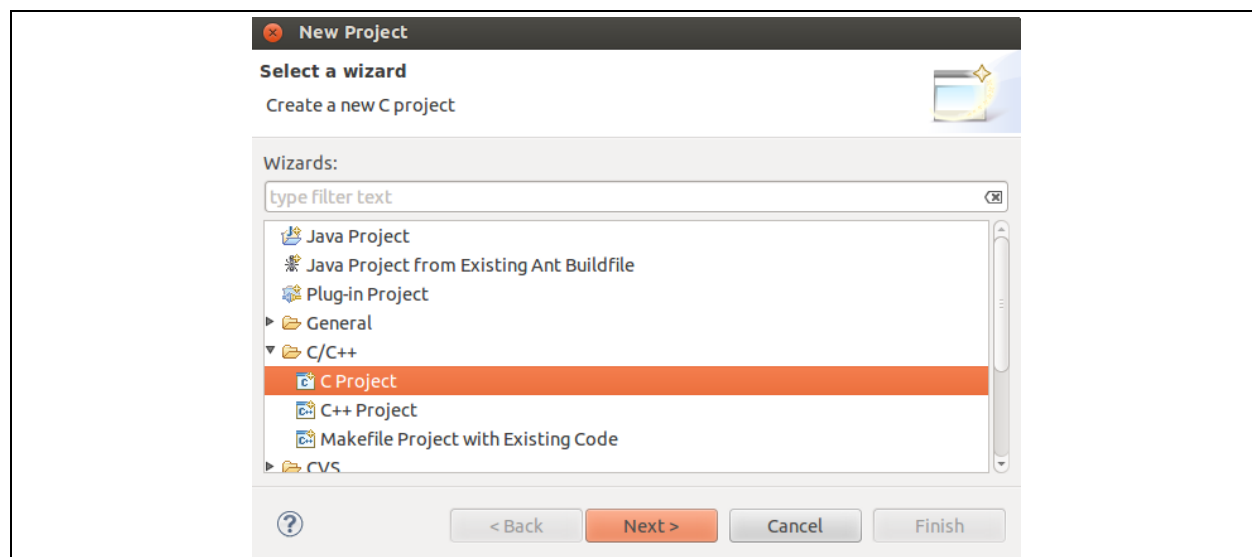


3.5 Creating a new project on Eclipse

After you set up the Edison toolchain, it is easy to create a new project with Eclipse:

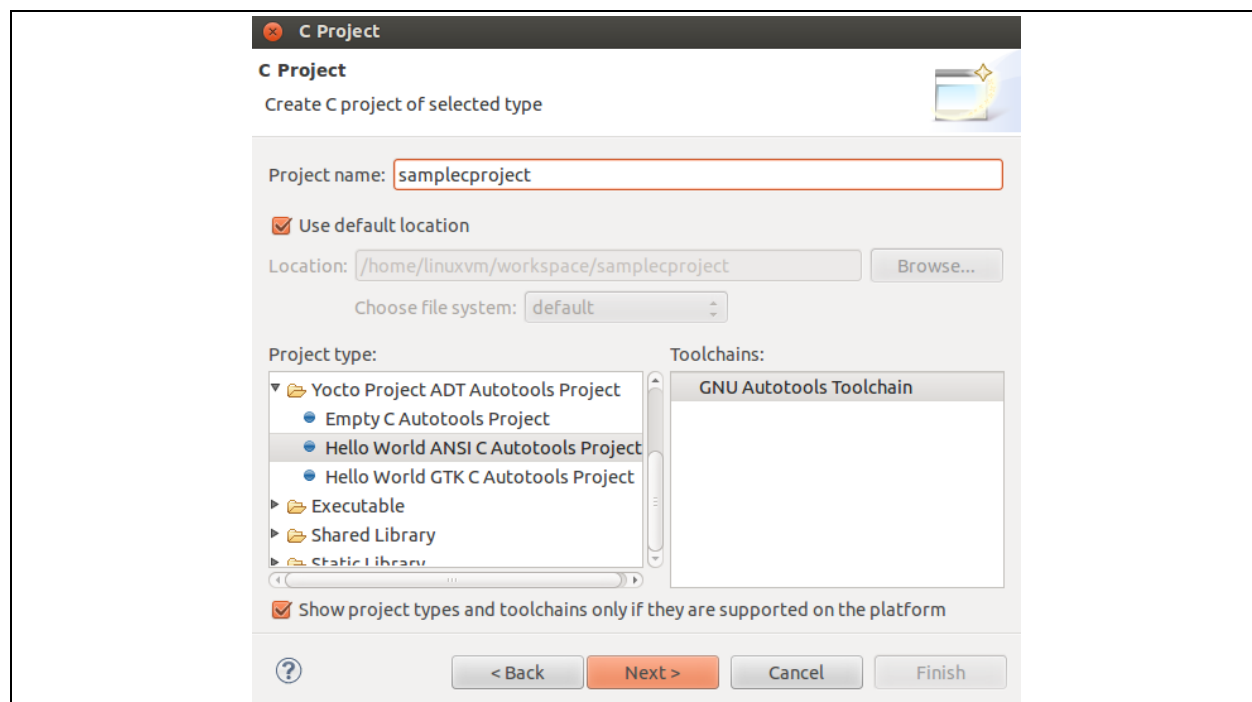
9. Select *File > New > Project*.
10. Select a C or C++ project from the C/C++ folder and click *Next* (Figure 12).

Figure 12 Select a new project



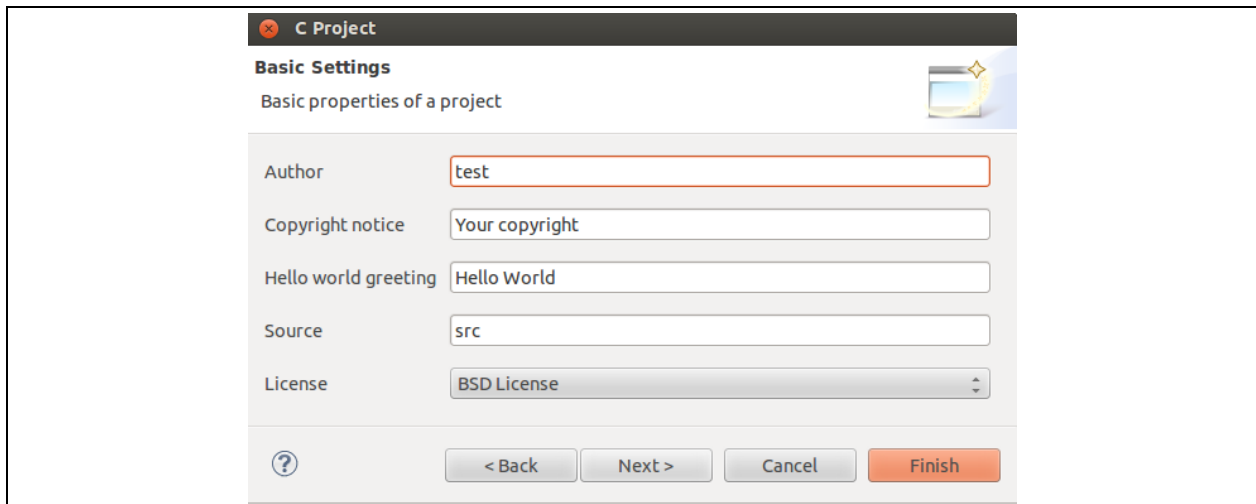
11. Enter a project name and select *Hello World ANSI C Autotools Project*. In this example (Figure 13), we selected a C project. If you select a C++ project, the project type will be C++.

Figure 13 A new C project



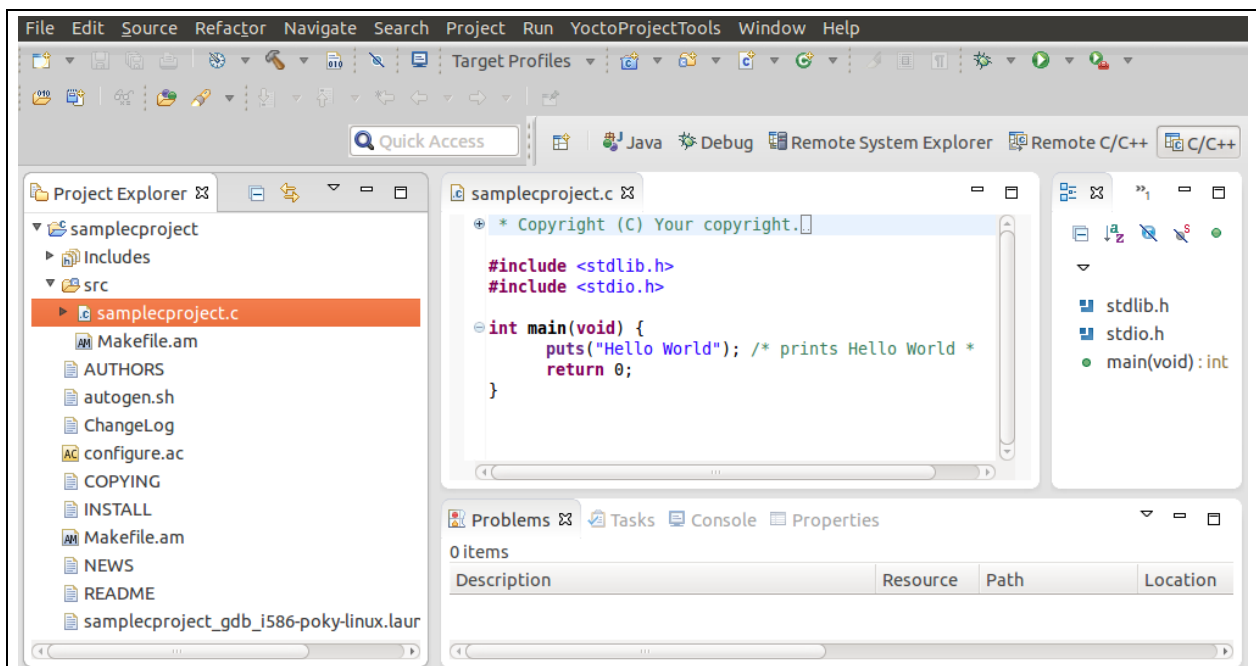
12. Fill in the fields for author, license, and copyright for your project (Figure 14).

Figure 14 C project basic settings



13. In the next window, enter the defined toolchain and you are ready to start your new project.

Figure 15 Defined toolchain for the new project

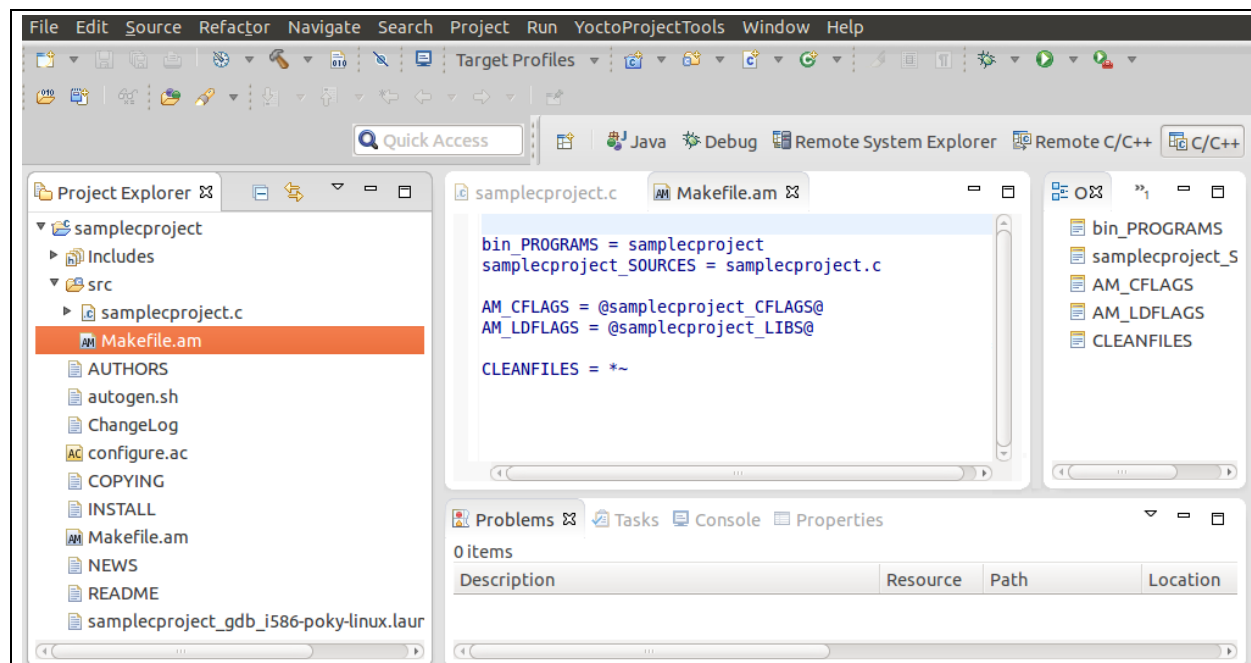


3.6 Development process on Eclipse

As soon as we created the *hello world* project in Section 3.5, it was ready to build and run on the target. Adding new code to the *.c or *.cpp file, or a new header or source files, is also quick and easy. Eclipse provides a text editor to implement C/C++ code because it recognizes compile time errors instantaneously. Autotools makes it a simple click to compile and build an application.

To make changes and add new source files, you can create new folders and change the *Makefile.am* file to build them together. A review of *Makefile.am* and folder structure is a good start. Figure 16 shows the *Makefile.am* file and the project folder's structure.

Figure 16 Project folder structure



3.7 Deployment with Eclipse

To run your application automatically on the remote Edison device, you must configure the project by doing the following:

14. Right-click on the project and select *Run As > Run Configurations* (Figure 17).

Figure 17 Configure project

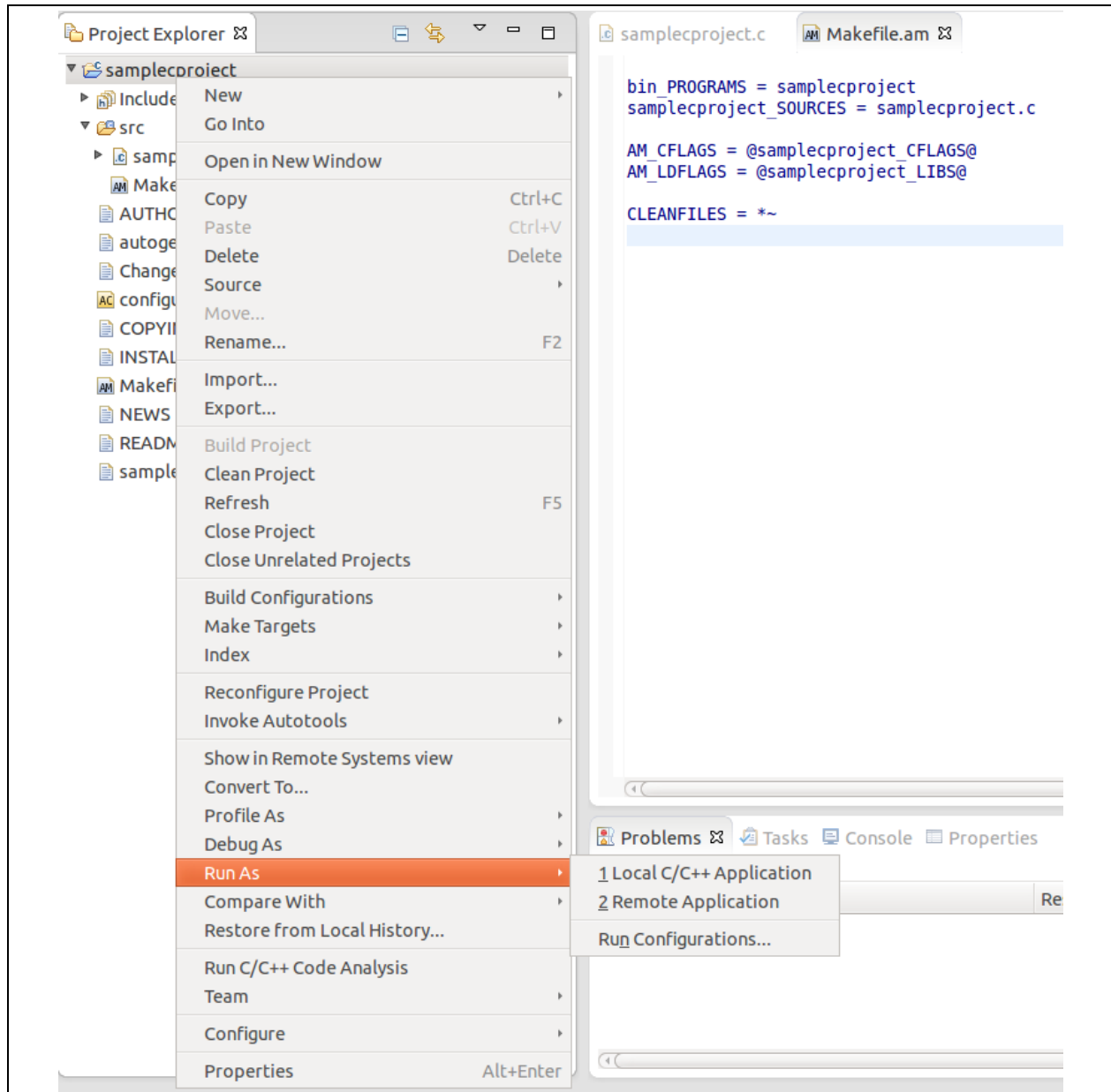
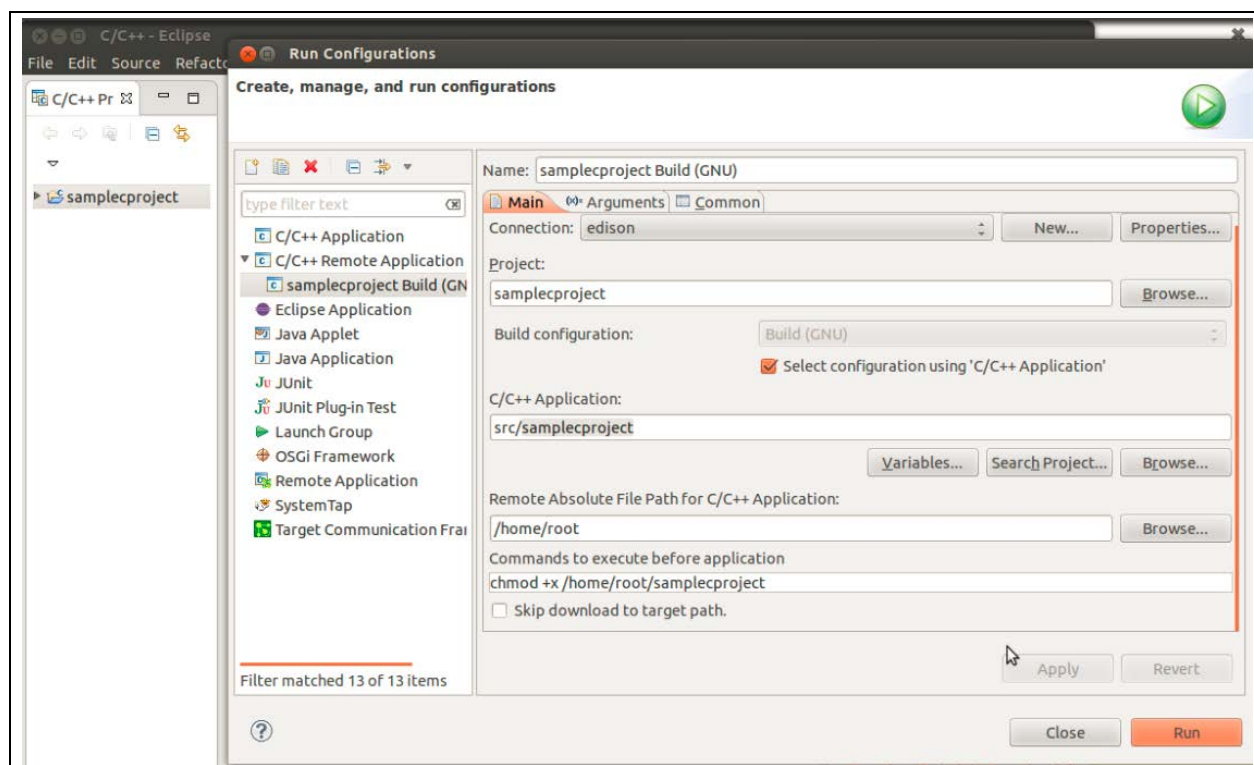


Figure 18 Run configurations



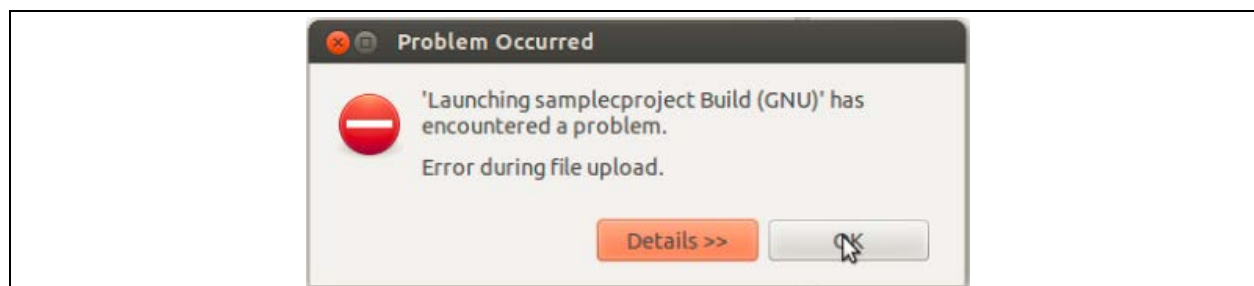
15. In the *Run configurations* window (Figure 18), complete the following:

- Double-click *C/C++ Remote Applications*, then select your build configuration, (*sampleproject Build (GNU)* in this example).
- On the *Main* tab, from the *Connection* dropdown menu, select the connection you created (in this case *edison*).
- In the *C/C++ Application* field, enter the project path/filename. You can click *Search project* or *Browse* to search for projects.
- In the *Remote Absolute File Path for C/C++ Application* field, enter the path where you want to deploy your application. You can also click *Browse* to see the folder structure on the target device.
- If your application has dependencies or if you need to change ownership of the created path, you can execute a command before running the application on the target device by entering the command in the *Commands to execute before application* field at the bottom.

16. Click *Run*.

17. If there is a problem running the project, you might see an error message (Figure 19).

Figure 19 Problem occurred



18. Go to the absolute file path in the Intel® Edison device's shell where the binaries have been deployed and change the permission of the binaries to executable, then run the binary from the Intel® Edison device's shell (Figure 20).

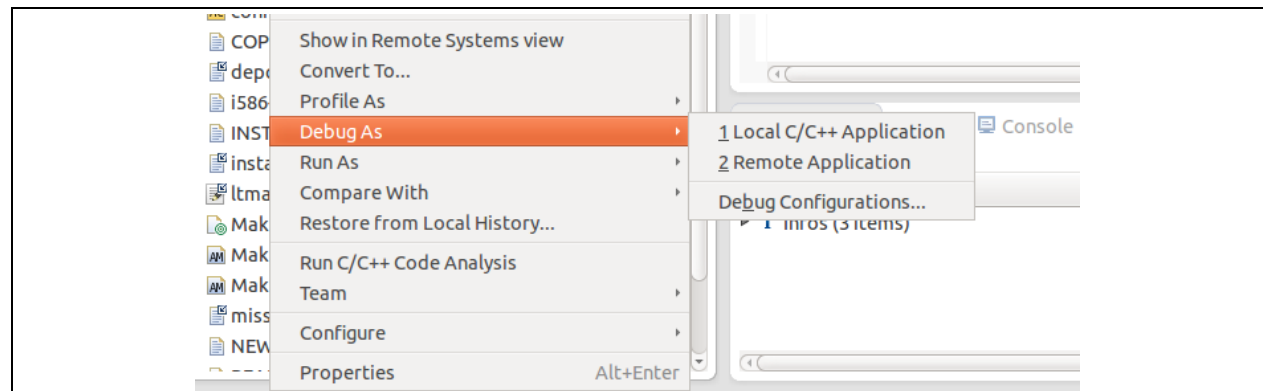
Figure 20 Run the binary from the Intel® Edison device's shell

```
root@edison:~# pwd
/home/root
root@edison:~# ls
sampleproject
root@edison:~#
root@edison:~# chmod +x sampleproject
root@edison:~#
root@edison:~# ./sampleproject
Hello World
root@edison:~# █
```

3.8 Debugging with Eclipse

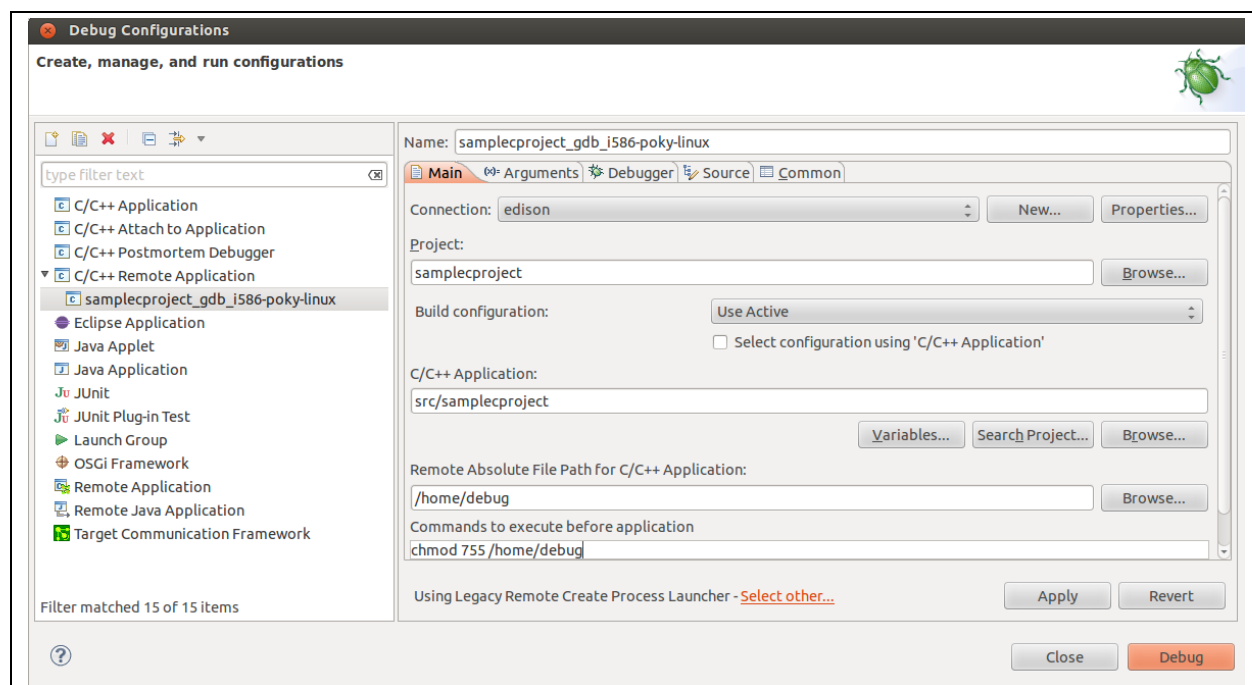
Debugging a configuration is similar to running a configuration (previous section), but instead of selecting *Run As > Run Configurations*, select *Debug As > Debug Configurations* (Figure 21).

Figure 21 Debug configurations



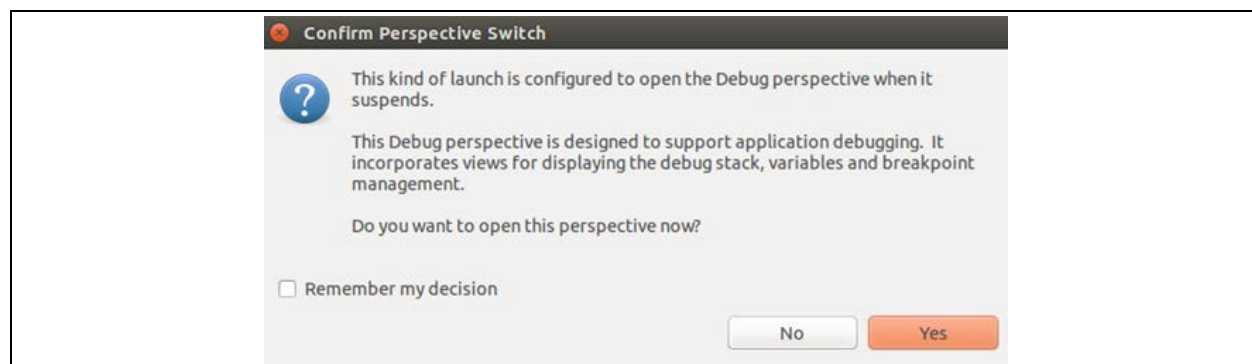
After you complete the configuration, you can start debugging by clicking *Debug* (Figure 22).

Figure 22 Debug



If this is the first time you have used this debugger, it will ask you to confirm the perspective switch (Figure 23).

Figure 23 Confirm perspective switch



After confirmation, Eclipse will change to Debug perspective.

§



4 Porting an existing project to Intel® Edison

In many cases, you may have an existing project that already runs on a different platform and you want to run it on the Edison Development Platform. To use an existing project on the Edison Development Platform, you will need to compile and build the source code of the library with the Edison toolchain.

Note: This example uses the *libjpeg* open source library, a widely used free software library written in C that implements JPEG encoding/decoding functions alongside various utilities for handling JPEG images.

To start the porting process, download the *libjpeg* source code from the official website (<http://ijg.org/files>) and untar the source to your working directory:

```
wget http://ijg.org/files/jpegsrc.v9.tar.gz
tar xzf jpegsrc.v9.tar.gz
jpeg-9/
jpeg-9/wrppm.c
jpeg-9/wrrle.c
jpeg-9/maketdsp.vc6
jpeg-9/jdinput.c
jpeg-9/testimg.bmp
jpeg-9/testimg.jpg .....
```

Because *libjpeg* is an Autotools project, it is fairly easy to configure for a build. Before starting for the build, set the Edison toolchain environment variables as described in Section 2.3, configure the host environment, and change directories to the extract folder *jpeg-9*.

```
source /opt/poky-edison/1.5.1/environment-setup-i586-poky-linux
cd jpeg-9
```

Since environment setup for Edison will configure \$CC and \$LD variables, running the *configure* script in the source folder will configure the build variables for *libjpeg*.

```
./configure
configure: loading site script /opt/poky-edison/1.5.1/site-config-i586-poky-linux
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking whether make supports nested variables... (cached) yes
checking whether to enable maintainer-specific portions of Makefiles... no
checking for gcc... i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky-edison/1.5.1/sysroots/i586-poky-linux .....
...
make
make[1]: Entering directory `/home/linuxvm/Templates/jpeg-9'
CC      jaricom.lo
CC      jcapimin.lo
CC      jcapistd.lo
CC      jcarith.lo
CC      jccoefct.lo
CC      jccolor.lo
CC      jcdctmgr.lo
CC      jchuff.lo .....
```



The *configure* script will create the necessary files for the build. Then you can run *make* as shown above to create binaries and libraries. After you build the *libjpeg* with the Edison toolchain, the binaries are ready to use and deploy to the file system on board.

4.1 Using external libraries

Porting *libjpeg* is simpler than porting a more complex project with many dependencies. With mainstream Linux distributions like Ubuntu, you can get external dependencies via *apt-get*, if the library exists in a defined repository.

If a project that you want to run needs external libraries that are not included in the Edison toolchain, you can port the library with the method described in Section 0 and statically link headers and library to the project environment. For example, we have a basic project that needs to use *libjpeg*. To include the header and link the ported library for the example project, use the gcc flags *-I* and *-L*:

```
$CC -o jpegSample jpegtest.c -I/path/to/jpeg-9 -L/path/to/libJpeg -ljpeg
```





5 Simple Native Applications

This section explains how to build a native application in Windows and Linux environments.

5.1 Windows native applications

To build a native application in a Windows environment, do the following:

1. Unzip the *edison-sdk-win32-weekly-14.zip* file to a directory of your choice.

```
/* Hello World program */
#include<stdio.h>
int main(void){
main()
{
printf("Hello World\n");
}
```

2. If you extract the zip archive to the C:\ drive, the command to build *helloworld.c* would be this:

```
C:\edison-sdk-win32-weekly-14\poky-edison-eglibc-i686-edison-image-core2-32-
toolchain-1.6\sysroots\i686-pokysdk-mingw32\usr\bin\i586-poky-linux\i586-poky-
linux-gcc.exe --sysroot=C:\edison-sdk-win32-weekly-14\poky-edison-eglibc-i686-
edison-image-core2-32-toolchain-1.6\sysroots\core2-32-poky-linux
c:\test\helloworld.c -o c:\test\helloworld
```

5.2 Linux native applications

To build a native application for the target using the cross-compilation toolchain, do the following:

3. Install the cross-compiler in */opt/poky*:

```
sudo ./tmp/deploy/sdk/poky-eglibc-x86_64-edison-image-i586-toolchain-1.5.1.sh
```

4. Initialize the environment to use the proper cross-compiler:

```
source /opt/poky/1.6/environment-setup-i586-poky-linux
```

5. Build a "helloworld" C program:

```
/* Hello World program */
#include<stdio.h>
main()
{
printf("Hello World\n");
}
```

6. Save it as *helloworld.c*.

7. Compile the *helloworld.c* program and deploy it on the device:

```
CC -o helloworld helloworld.c
scp helloworld root@192.168.2.15:/home/root
```



5.3 Sample GPIO Write Application

The `write_gpio_pin.c` sample code is listed below. This sample code changes GPIO pin 17's direction to OUT and sets its value as HIGH.

```
/* A Sample Program to set GPIO pin 17 direction OUT and value as HIGH
**
** Author: Onur Dundar
**
*/

#include <stdio.h>
#include <fcntl.h>
#define GPIO_DIRECTION_PATH "/sys/class/gpio/gpio%d/direction"
#define GPIO_VALUE_PATH "/sys/class/gpio/gpio%d/value"
#define GPIO_EXPORT_PATH "/sys/class/gpio/export"
#define BUFFER 50

int main()
{
//GPIO Pin 17
    int gpio_pin = 17;
//Path Variables
    char gpio_exp_path[BUFFER];
    char gpio_direction_path[BUFFER];
    char gpio_value_path[BUFFER];
//Files
    int fd_export, fd_val, fd_dir;
    int err = 0;
//Set GPIO Paths

    snprintf(gpio_exp_path, BUFFER, GPIO_EXPORT_PATH, gpio_pin);
    snprintf(gpio_direction_path, BUFFER, GPIO_DIRECTION_PATH, gpio_pin);
    snprintf(gpio_value_path, BUFFER, GPIO_VALUE_PATH, gpio_pin);

//Set Direction
    fd_dir = open(gpio_direction_path, O_WRONLY);
    if (fd_dir < 0) {
        perror("Can't Open GPIO Direction File");
//Export GPIO
        fd_export = open(gpio_exp_path, O_WRONLY);
        if (fd_export < 0) {
            perror("Can't Open Export File");
            return -1;
        } else {
//Export GPIO
            char buf[15];
            sprintf(buf, "%d", gpio_pin);
            err = write(fd_export, buf, sizeof(buf));
            if (err == -1) {
                perror("Can't export GPIO");
                return err;
            }
            close(fd_dir);

            fd_dir = open(gpio_direction_path, O_WRONLY);
            if (fd_dir < 0) {
                perror
                    ("Can't Open Exported GPIO Direction File");
                return -1;
            }
        }
    }
}
```



```
    }

    fd_val = open(gpio_value_path, O_WRONLY);
    if (fd_val < 0) {
        perror("Can't Open GPIO Value File");
        return -1;
    }
//Now Continue to Set Direction as out
    err = write(fd_dir, "out", sizeof("out"));
    if (err == -1) {
        perror("Can't set direction");
        return err;
    } else {
        printf("Set gpio %d direction as out\n", gpio_pin);
    }

    err = write(fd_val, "1", sizeof("1"));
    if (err == -1) {
        perror("Can't set value");
        return err;
    } else {
        printf("Set gpio %d value as HIGH\n", gpio_pin);
    }

// Close the Files
    close(fd_dir);
    close(fd_val);

    return 0;
}
/** End of GPIO Program **/
```

§

6 Sample Pedometer Application

This chapter shows how to implement a native software application for a pedometer that counts a person's steps. It uses an Intel® Edison Development Board and an Invensense* MPU6050* 6-axis accelerometer/gyroscope (Figure 24).

Figure 24 Hardware required for the pedometer application

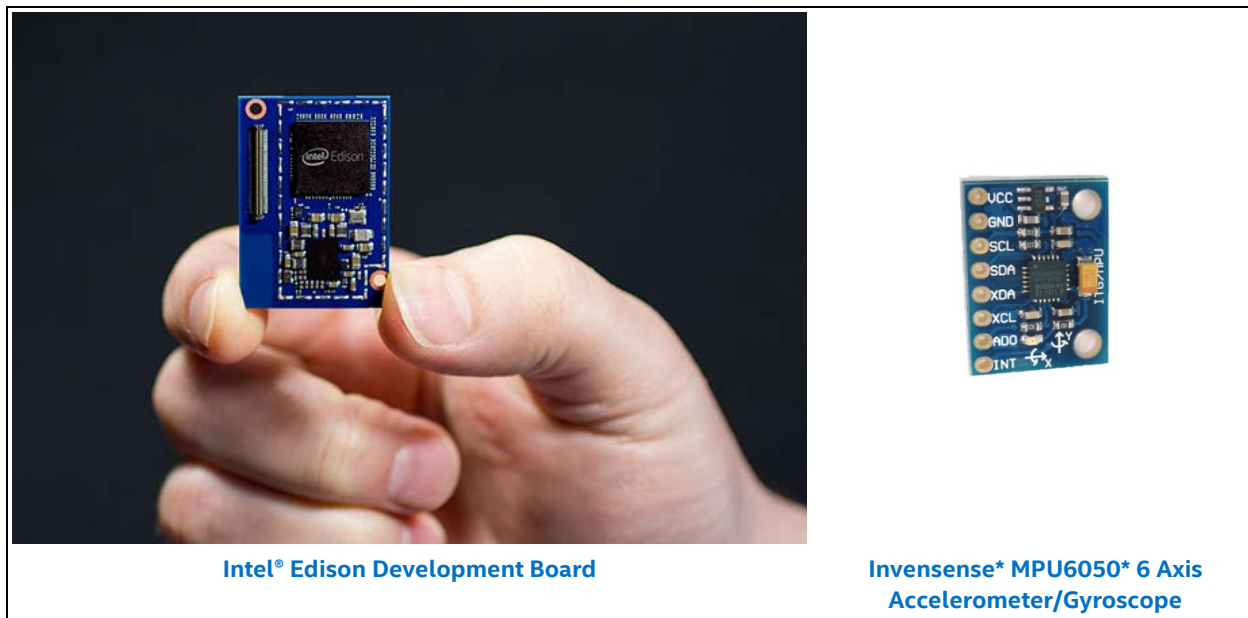
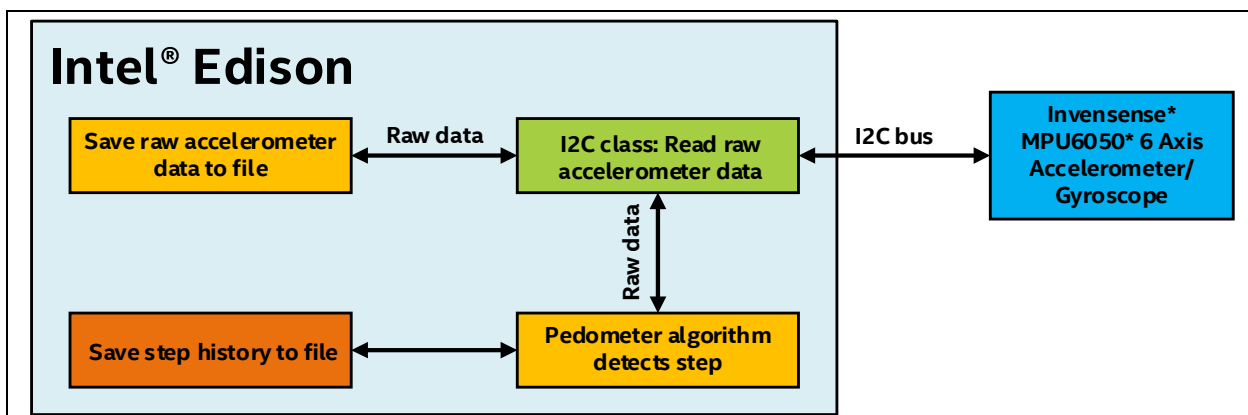


Figure 25 shows the software block diagram for integrating the hardware for this application.

Figure 25 Software block diagram



Before developing this application, configure your host environment or configure Eclipse to use the Edison toolchain. (See section 2 for details.)

This sample project includes three source files named `pedometer.c` (includes 'main' function), `i2c.c` and `i2c.h` to define a simple i2c library to use for I²C communication.



6.1 Reading accelerometer raw data

The Edison device connects to the MPU6050 device via I²C, to read the MPU6050's registers. The Linux kernel provides an *i2c-dev* library to communicate with I²C devices from the user space, accessing I²C char device interface in the */dev* directory, in this case *i2c-0*. For details about I²C, refer to the I²C documentation listed in section 1.1.

6.1.1 I²C operations

Because we need to access MPU6050 via I²C, some basic functions to open I²C adapter, setting I²C slave, read and write registers have been implemented in *i2c* class in this sample. Because the *i2c-dev* library has SMBus protocol functions for read and write operations that are not included in the toolchain, we must implement these functions as inline functions in an *i2c.h* file. Also note that we have copied SMBus functions from the Intel® Galileo libraries into the *../hardware/arduino/x86/cores/arduino/i2c-dev.h* file.

Here is a list of SMBus functions used in *i2c.h*:

```
static inline __s32 i2c_smbus_access(int file, char read_write, __u8 command,
                                     int size, union i2c_smbus_data *data)
static inline __s32 i2c_smbus_read_byte(int file)
static inline __s32 i2c_smbus_write_byte(int file, __u8 value)
static inline __s32 i2c_smbus_read_byte_data(int file, __u8 command)
static inline __s32 i2c_smbus_write_byte_data(int file, __u8 command, __u8 value)
static inline __s32 i2c_smbus_read_block_data(int file, __u8 command, __u8 *values)
static inline __s32 i2c_smbus_write_block_data(int file, __u8 command,
                                                __u8 length, const __u8 *values)
static inline __s32 i2c_smbus_block_process_call(int file, __u8 command, __u8 length,
__u8 *values)
```

Here is a list of I²C functions used. These functions were implemented with SMBus protocol functions:

```
//Open i2c adapter file
int i2c_open_device(int adapter_num)
//Set i2c slave address
int i2c_set_slave(int dev_file, int dev_addr)
//Read
int i2c_receive_byte(int dev_file);
int i2c_read_byte(int dev_file, uint8_t regaddr)
int i2c_read_bytes(int dev_file, uint8_t *buf, uint8_t length, uint8_t regaddr)
//Write
int i2c_send_byte(int dev_file, uint8_t value)
int i2c_write_byte(int dev_file, uint8_t regaddr, uint8_t value)
int i2c_write_bytes(int dev_file, uint8_t *bytes, uint8_t length)
```

Before starting I²C read and write operations, we must connect with I²C devices from the user space. To connect with device, I²C adapter file should be accessed. *i2c_open_device* function includes the I²C file open operation. The adapter number is passed as a parameter.

```
int i2c_open_device(int adapter_num) {
    char buf[MAX_BUF];
    if(snprintf(buf, sizeof(buf), "/dev/i2c-%d", adapter_num)>0){
        perror("Can't create adapter path\n");
    }
    int dev_file = open(buf, O_RDWR);
    if (dev_file < 0) {
        perror("Failed to open adapter\n");
        return -1;
    }
    return dev_file;
}
```




After accessing I²C device file, we have to set the slave address for the I²C device to connect and perform read and write operations from device through I2C adapter.

```
int i2c_set_slave(int dev_file, int dev_addr) {
    if (ioctl(dev_file, I2C_SLAVE_FORCE, dev_addr) < 0) {
        puts("Failed to acquire bus access and/or talk to slave.\n");
        return -1;
    }
    return 0;
}
```

After accessing the I²C char device file and set slave bus address, read and write operations can occur. Read and write functions have been developed with previously defined SMBus functions.

```
//Receives Bytes from i2c device
int i2c_receive_byte(int dev_file) {
    int byte;
    if ((byte = i2c_smbus_read_byte(dev_file)) < 0) {
        perror("Failed to receive byte from I2C slave\n");
        return -1;
    }
    return byte;
}

// Send Byte to i2c device
int i2c_send_byte(int dev_file, uint8_t value) {
    if (i2c_smbus_write_byte(dev_file, value) < 0) {
        perror("Failed to write byte to I2C slave\n");
        return -1;
    }
    return 0;
}

//Read a chunk of Bytes from device registers
int i2c_read_bytes(int dev_file, uint8_t *buf, uint8_t length, uint8_t regaddr) {
    int ret;
    if ((ret = i2c_smbus_read_i2c_block_data(dev_file, regaddr, length, buf))
        < 0) {
        perror("Failed to read bytes from I2C slave\n");
        return -1;
    }
    return ret;
}

//Write a chunk of bytes to device registers
int i2c_write_bytes(int dev_file, uint8_t *bytes, uint8_t length) {
    if (i2c_smbus_write_i2c_block_data(dev_file, bytes[0], length - 1,
        bytes + 1) < 0) {
        perror("Failed to write bytes to I2C slave\n");
        return -1;
    }
    return 0;
}
```

Above I²C functions are enough for this sample to access, read, and write MPU6050 registers, even though in the main function read and write operation has been completed with below functions in one step. To read from the device's specific register, the register address needs to be sent to the device, and then the slave should be reset for the read process. MPU6050 functions are also defined to work with bit operation functions, which call below-byte read/write functions in the MPU6050 library. Implementing these functions would make the porting process easier.

```
//I2C Read and Write Operations in main function
int writeByte(uint8_t regAddr, uint8_t* data);
int writeBytes(uint8_t regAddr, uint8_t length, uint8_t* data);
int readByte(uint8_t regAddr, uint8_t *data);
int readBytes(uint8_t regAddr, uint8_t length, uint8_t *data);
```



The *i2cdev* library provides *writeBit*, *writeBits*, and *readBits* functions (below). MPU6050 functions directly calls these functions to perform I²C operations.

```
//Bit Operations
int8_t readBits(uint8_t regAddr, uint8_t bitStart, uint8_t length, uint8_t *data);
int writeBits(uint8_t regAddr, uint8_t bitStart, uint8_t length, uint8_t data);
int writeBit( uint8_t regAddr, uint8_t bitNum, uint8_t data);
```

6.1.2 Communication with MPU6050

Getting raw accelerometer data from an MPU6050 device is implemented with the *main* function. To get the values, you will need the above I²C functions by MPU6050 functions. The first step is to access the I²C adapter.

```
int i2c_device_address; i2c_adapter;
...//some definitions here
int main(int argc, char *argv[]){
    ...//Some code here
    i2c_device_address = MPU6050_DEFAULT_ADDRESS;
    i2c_adapter = i2c_open_device(I2CAdapter);
    if (i2c_adapter < 0) {
        printf("Can't Open i2c Adapter\n");
        return -1;
    }...//Some code here
```

Global variables to define MPU6050 registers and required device command values have been defined. Other global variables are defined to store accelerometer values, device file, and so on.

```
#define I2CAdapter 0
#define BUFFER_LENGTH 32
//Accelerometer Values
int16_t ax, ay, az;
//Gyroscope Values
int16_t gx, gy, gz;
uint8_t buffer[14];
//MPU6050 vars
#define MPU6050_ADDRESS_AD0_LOW 0x68 // address pin low (GND), default for
InvenSense evaluation board
#define MPU6050_ADDRESS_AD0_HIGH 0x69 // address pin high (VCC)
#define MPU6050_DEFAULT_ADDRESS MPU6050_ADDRESS_AD0_LOW
//Bit and Byte Macros
#define MPU6050_RA_PWR_MGMT_1 0x6B
#define MPU6050_PWR1_CLKSEL_BIT 2
#define MPU6050_PWR1_CLKSEL_LENGTH 3
//Initialization
#define MPU6050_CLOCK_PLL_XGYRO 0x01
#define MPU6050_GYRO_FS_250 0x00
#define MPU6050_ACCEL_FS_2 0x00
//MPU6050 configurations
#define MPU6050_RA_ACCEL_CONFIG 0x1C
#define MPU6050_ACONFIG_AFS_SEL_BIT 4
#define MPU6050_ACONFIG_AFS_SEL_LENGTH 2
#define MPU6050_RA_GYRO_CONFIG 0x1B
#define MPU6050_GCONFIG_FS_SEL_BIT 4
#define MPU6050_GCONFIG_FS_SEL_LENGTH 2
#define MPU6050_RA_PWR_MGMT_1 0x6B
#define MPU6050_PWR1_SLEEP_BIT 6
#define MPU6050_RA_WHO_AM_I 0x75
#define MPU6050_WHO_AM_I_BIT 6
#define MPU6050_WHO_AM_I_LENGTH 6
#define MPU6050_RA_ACCEL_XOUT_H 0x3B
```



Note: MPU6050-related operations and constant variables reference the MPU6050.cpp, MPU6050.h, I2Cdev.cpp, and I2Cdev.h libraries from <https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050>. Only necessary functions and global variables are used from these libraries to keep development simple. It is also possible to use the MPU6050 and I²Cdev libraries within the project.

Functions to use from the MPU6050 and i2cdev libraries:

```
//MPU6050 Functions
//MPU6050 Initialization
void setClockSource(uint8_t source);
void setFullScaleGyroRange(uint8_t range);
void setSleepEnabled(int enabled);
//Get Raw Data
void getMotion6(int16_t* ax, int16_t* ay, int16_t* az, int16_t* gx, int16_t*
gy, int16_t* gz);
void getAcceleration(int16_t* x, int16_t* y, int16_t* z);
//Connection Test
uint8_t getDeviceID();
int testConnection();
```

Device initialization needs to call functions to set the clock source and the accelerometer range, and to disable the sleep function (to get continuous data).

```
..// main function
// Initialize MPU6050
printf("Initializing MPU6050 devices...\n");
setClockSource(MPU6050_CLOCK_PLL_XGYRO);
setFullScaleGyroRange(MPU6050_GYRO_FS_250);
setFullScaleAccelRange(MPU6050_ACCEL_FS_2);
setSleepEnabled(0);
//... Some code here
```

Next we have to verify that the device is connected to the MPU6050 by reading the MPU6050's device ID, a required step to verify connection. Connection test functions.

```
..// main function
// Test Connection
printf("Testing device connections...\n");
printf(testConnection() ? "MPU6050 connection successful\n" : "MPU6050
connection failed\n");
//... Some code here
```

The application is ready to read data from the MPU6050.



6.1.3 Reading raw data

When above operations successfully developed, the device is ready to provide raw accelerometer data. In order to do that, below functions has been defined in the MPU6050 library.

```
void getMotion6(int16_t* ax, int16_t* ay, int16_t* az, int16_t* gx, int16_t*
gy, int16_t* gz) {
    readBytes(MPU6050_RA_ACCEL_XOUT_H, 14, buffer);
    *ax = (((int16_t) buffer[0]) << 8) | buffer[1];
    *ay = (((int16_t) buffer[2]) << 8) | buffer[3];
    *az = (((int16_t) buffer[4]) << 8) | buffer[5];
    *gx = (((int16_t) buffer[8]) << 8) | buffer[9];
    *gy = (((int16_t) buffer[10]) << 8) | buffer[11];
    *gz = (((int16_t) buffer[12]) << 8) | buffer[13];
}

void getAcceleration(int16_t* x, int16_t* y, int16_t* z) {
    readBytes(MPU6050_RA_ACCEL_XOUT_H, 6, buffer);
    *x = (((int16_t) buffer[0]) << 8) | buffer[1];
    *y = (((int16_t) buffer[2]) << 8) | buffer[3];
    *z = (((int16_t) buffer[4]) << 8) | buffer[5];
}
```

6.1.4 Pedometer algorithm

In the previous section, raw values have been read from MPU6050. All we need to do is to provide raw data to the algorithm to detect the user's steps and tally them.

```
int main(int argc, char *argv[]){
....
...
    y_offset = 240;
    //gravity mean of the read z-axis raw data from MPU6050
    scale[0] = -(int16_t) (480 * 0.5f * (1.0f / GRAVITY));

    //magnetic field check if required 269
    scale[1] = -(int16_t) (480 * 0.5f * (1.0f / (60)));
    int i = 0;
    for (; i < 6; i++) {
        lastValues[i] = 0;
        lastDirections[i] = 0;
    }
    for (;;) {
        vSum = 0;
        int16_t v;
        /**
         * Poll Accel Raw Data
         */
        getAcceleration(&ax, &ay, &az);
        printf("accel: %d\t%d\t%d\t\n", ax, ay, az);
        /**
         * Vector Sum Calculation
         */
        vSum = (ax * scale[1] + y_offset) + (ay * scale[1] + y_offset)
              + (az * scale[1] + y_offset);
        v = vSum / 2300;
        int k = 0;
        /**
         * Detect Direction Change
         */
        int16_t direction = (
            v > lastValues[k] ? 1 : (v < lastValues[k] ? -1 : 0));
        if (direction == -lastDirections[k]) {
```



```

        // Check if Direction changed

        int etype = (direction > 0 ? 0 : 1); // minumum or maximum?
        lastExtremes[etype][k] = lastValues[k];
        int16_t diff = abs(
            lastExtremes[etype][k] - lastExtremes[1 -
etype][k]);

        if (diff > limit) {

            // if is almost as large as prev and is previous is
large enough

            if ((diff > ((lastDiff[k] * 99) / 100))
                && (lastDiff[k] > (diff / 100))
                && (lastMatch != 1 - etype)) {
                stepCount++;
                printf("Step Count: %d\n", stepCount);
                lastMatch = etype;
            }

            else {

                lastMatch = -1;
            }
        }

        lastDiff[k] = diff;
        lastDirections[k] = direction;
        lastValues[k] = v;
        usleep(200000);
    }

...
return 0;
}

```



6.2 Saving data and distance, calorie calculation

Detecting step is the main course of this sample but more can be done with the counted step data. In this sample application, counted steps have been saved and number of steps used to calculate approximate distances walked and calories burned.

Data saved in xml format to make it easy to be read by NodeJS service for web visualization.

```
...//some code here
file_ptr = fopen("data.xml","w+");
    if(!file_ptr){
        perror("Can't open file");
    }
    clock_t timestamp = clock();
    /**
     * Write Data with XML Format
     */
    fprintf(file_ptr,"<data>\n");
    fprintf(file_ptr,"<timestamp>%f</timestamp>\n", (float)timestamp);
    fprintf(file_ptr,"<x>%d</x>\n",ax);
    fprintf(file_ptr,"<y>%d</y>\n",ay);
    fprintf(file_ptr,"<z>%d</z>\n",az);
    fprintf(file_ptr,"<vector>%d</vector>\n",v);
    fprintf(file_ptr,"<count>%d</count>\n",stepCount);
    fprintf(file_ptr,"<distance>%f</distance>\n",distance);
    fprintf(file_ptr,"<calorie>%f</calorie>\n",calories_burnt);
    fprintf(file_ptr,"</data>\n");
    fclose(file_ptr);
.....//some code here
```

We define the length of each step as 0.5 (meters) and calculate the distance walked using this value.

```
#define DistancePerStep 0.5
float distance = 0.0;
//at each step ...//some code

Distance += DistancePerStep; // some code here
```

To calculate burned calories, we refer to an article from **livestrong.com** (<http://www.livestrong.com/article/238020-how-to-convert-pedometer-steps-to-calories>), which calculates calories burned for each mile. In the sample application, we altered this formula to approximate the calories expended for each step and multiplied this by the number of steps to show the total number of calories burned. In this sample, the user's weight is hardcoded as 80 kilograms.

```
#define Weight 80.0
#define CaloriePerStep (((0.57)*(2.20462)*Weight)/(3218.69))
float calories_burnt = 0.0;
..// At each step
calories_burnt += CaloriePerStep; //some code here
```

§