

EMBSYS 110 Module 1

Design and Optimization of Embedded and Real-time Systems

1 Introduction to Software Design

1.1 My First Job

When I first came out to work as an embedded software engineer, a senior told me "Your job of programming is simple. You just need to tell the micro-controller to do what you want it to do."

Obviously it is an over-simplification. It does, however, brings up a couple interesting questions:

1. Do we know ourselves what we want to computer to do? How are we certain that we really know it? Is our knowledge complete and free of contradictions? How can we capture this knowledge?
2. Assuming the answer to the first question above is "yes", how do we transfer this knowledge to the computer in a precise and straight-forward manner?

In the age of AI and machine learning, these might not be the right questions to ask anymore. The questions might be inverted to become "How can a computer tell us what it wants to do?"

Indeed in the fields of safety-critical systems, AI still presents many challenges. For the time being, our questions above are still relevant, and lets take a deeper look at them.

1.2 Two Key Concepts

The first set of questions are about *knowledge representation*. They are very tricky since knowledge is very abstract. Human thoughts are very flexible but often are not precise enough. We do better in reasoning about sequential logic (like if ... then, do ... while, wait until ... etc.) than reasoning about things that can happen in parallel.

The knowledge we are talking about here boils down to system behaviors. Since human minds aren't very good intuitively to reason about it, we need a tool to help us, a tool that helps us define behaviors, capture it and visualize it.

It is such a difficult problem that it took a computer scientist to find a solution. It was David Harel who invented statecharts back in late 1980s to solve this exact problem – how to specify system behaviors for embedded systems (a fighter jet in his case). We will look into the rather interesting history behind it later.

Now let's address the second question – how do we transfer our knowledge to the computer in a precise and straight-forward manner? The short answer is of course by programming. However as anyone with some programming experience will tell, if you simply program your sequential thoughts into code it won't take long for your program to evolve into a big messy ball of spaghetti. This is a *mapping* problem. We need an effective tool to map system behaviors (specification or knowledge) into executable code. Such tools pretty much has existed as long as statecharts. However they are very expensive. It is not until the advent of Quantum Framework/Platform by Miro Samek in 2000 that made it truly affordable and practical to everyone.

Now we attempt to answer the question of "What is software design?"

Software design is

1. A complete and precise specification of system behaviors that fulfills the requirements.

We call it the *behavioral model* which is the representation of our knowledge about what the system is required to do. Statecharts are simple yet powerful tools that help us achieve this.

2. A software architecture that supports the execution of the model.

Quantum Platform (QP) is one of such tools that help developers map or translate statecharts to C/C++ code effectively.

1.3 A Couple Myths

The first myth is that a design would automatically arise through constant integration, refactoring and testing. There is no doubt that *some* design would come up through the above process. It cannot be argued either that all of the above process are critical to a successful software project. However it is doubtful whether a good, reliable and maintainable design would implicitly come about without employing solid and dedicated design methods and tools.

We are fortunate as software developers nowadays as we have seen a proliferation of many great tools to help us with our development process. To name a few, they include:

1. git for source control and code review
2. cmake for configuration and build control
3. jenkins for build automation
4. unit test frameworks for unit testing
5. Agile for team and project management
6. Python for various automation.
7. static and dynamic code analysis tools.

As said all of the above are very useful tools. They may yield pieces of beautiful code but there is no guarantee that when working together they will produce a reliable and maintainable system as a whole. In other words, they are great tools that support our development process, but they do not replace the center piece of the picture which is **design**.

The second myth is that embedded systems were equivalent to resource constrained systems, i.e. those with limited computational power and memory. With the advent of low-cost yet powerful platforms such as Raspberry Pi or Beaglebone (GHz CPU with GB of memory), we no longer need to apply techniques specialized for embedded systems.

While it is true that many embedded systems are resource constrained, and many techniques we have learned focus on optimization, resource is *not* the defining property of embedded systems. As we shall see in the upcoming story, embedded systems are all about *control, behaviors, determinism and reliability*.

1.4 Story of Statecharts

This is the story of statecharts told by their inventor David Harel in his paper in:

Statecharts in the Making: A Personal Account. David Harel. ACM Communications. <issue>
(<http://www.wisdom.weizmann.ac.il/~harel/papers/Statecharts.History.pdf>)

Here are some of the interesting quotes from it:

December 1982 to mid 1983: The Avionics Motivation

We cut now to December 1982. At this point I had already been on the faculty of the Weizmann Institute of Science in Israel for two years. One day, the same Jonah Lavi called, asking if we could meet. In the meeting, he described briefly some severe problems that the engineers at IAI seemed to have, particularly mentioning the effort underway at IAI to build a home-made fighter aircraft...

And so, starting in December 1982, for several months, Thursday became my consulting day at the IAI. The first few weeks of this were devoted to sitting down with Jonah, in his office, trying to understand from him what the issues were.

*An avionics system is a great example of what Amir Pnueli and I later identified as a reactive system [HP85]. The aspect that dominates such a system is its **reactivity; its event-driven, control-driven, event-response nature, often including strict time constraints, and often exhibiting a great deal of parallelism**. A typical reactive system is not particularly data intensive or calculation-intensive. So what is/was the problem with such systems? In a nutshell, it is the need to provide a **clear yet precise description of what the system does, or should do. Specifying its behavior is the real issue**.*

Here is how the problem showed up in the Lavi. The avionics team had many amazingly talented experts. There were radar experts, flight control experts, electronic warfare experts, hardware experts, communication experts, software experts, etc. When the radar people were asked to talk about radar,

they would provide the exact algorithm the radar used in order to measure the distance to the target. The flight control people would talk about the synchronization between the controls in the cockpit and the flaps on the wings. The communications people would talk about the formatting of information traveling through the MuxBus communication line that runs lengthwise along the aircraft. And on and on. Each group had their own idiosyncratic way of thinking about the system, their own way of talking, their own diagrams, and their own emphases.

Then I would ask them what seemed like very simple specific questions, such as: "What happens when this button on the stick is pressed?" In way of responding, they would take out a two-volume document, written in structured natural language, each volume containing something like 900 or 1000 pages. In answer to the question above, they would open volume B on page 389, at clause 19.11.6.10, where it says that if you press this button, such and such a thing occurs... I would say: "Yes, but is that true even if there is an infra-red missile locked on a ground target?" To which they would respond: "Oh no, in volume A, on page 895, clause 6.12.3.7, it says that in such a case this other thing happens." This to-and-fro Q&A session often continued for a while, and by question number 5 or 6 they were often not sure of the answer and would call the customer for a response... By the time we got to question number 8 or 9 even those people often did not have an answer!

In my naïve eyes, this looked like a bizarre situation, because it was obvious that someone, eventually, would make a decision about what happens when you press a certain button under a certain set of circumstances. However, that person might very well turn out to be **a low-level programmer whose task it was to write some code for some procedure, and who inadvertently was making decisions that influenced crucial behavior on a much higher level.** Coming, as I did, from a clean-slate background in terms of avionics... this was shocking. It seemed extraordinary that this talented and professional team did have answers to questions such as "What algorithm is used by the radar to measure the distance to a target?", but in many cases did not have the answers to questions that seemed more basic, such as "What happens when you press this button on the stick under all possible circumstances?".

In retrospect, the two only real advantages I had over the avionics people were these: (i) having had no prior expertise or knowledge about this kind of system, which enabled me to approach it with a completely blank state of mind and think of it any which way; and (ii) having come from a slightly more mathematically rigorous background, making it somewhat more **difficult for them to convince me that a two-volume, 2000 page document, written in structured natural language, was a complete, comprehensive and consistent specification of the system's behavior.**

...let us take a look at an example taken from the specification of a certain chemical plant.

Section 2.7.6: Security "If the system sends a signal hot then send a message to the operator."

Section 9.3.4: Temperatures "If the system sends a signal hot and $T > 60^{\circ}$, then send a message to the operator."

Summary of critical aspects "When the temperature is maximum, the system should display a message

on the screen, unless no operator is on the site except when $T < 60^{\circ}$.”

Despite being educated as a logician, I've never really been able to figure out whether the third of these is equivalent to, or implies, any of the previous two... But that, of course, is not the point. The point is that these excerpts were obviously written by three different people for three different reasons, and that such large documents get handed over to programmers, some more experienced than others, to write the code. **It is almost certain that the person writing the code for this critical aspect of the chemical plant will produce something that will turn out to be problematic in the best case — catastrophic in the worst.** In addition, keep in mind that these excerpts were found by an extensive search through the entire document to try find where this little piece of behavior was actually mentioned. Imagine our programmer having to do that repeatedly for whatever parts of the system he/she is responsible for, and then to make sense of it all.

1983: Statecharts Emerging

The goal was to try to find, or to invent for these experts, a means for simply saying what they seemed to have had in their minds anyway. The goal was to find a way to help take the information that was present collectively in their heads and put it on paper, so to speak, in a fashion that was both well organized and accurate.

I became convinced from the start that the notion of a state and a transition to a new state was fundamental to their thinking about the system... They would repeatedly say things like, "When the aircraft is in air-ground mode and you press this button, it goes into air-air mode, but only if there is no radar locked on a ground target at the time". Of course, for anyone coming from computer science this is very familiar: what we really have here is a finite-state automaton, with its state transition table or state transition diagram. Still, it was pretty easy to see that just having one big state machine describing what is going on would be fruitless, and not only because of the number of states, which, of course, grows exponentially in the size of the system. Even more important seemed to be the pragmatic point of view, whereby **a formalism in which you simply list all possible states and specify all the transitions between them is unstructured and non-intuitive; it has no means for modularity, hiding of information, clustering, and separation of concerns**, and was not going to work for the kind of complex behavior in the avionics system. And if you tried to draw it visually you'd get spaghetti of the worst kind. It became obvious pretty quickly that it could be beneficial to come up with some kind of **structured and hierarchical extension of the conventional state machine formalism**.

...After a few of these meetings with the avionics experts, it suddenly dawned on me that everyone around the table seemed to understand the back-of-napkin style diagrams a lot better and related to them far more naturally. The pictures were simply doing a much better job of setting down on paper the system's behavior, as understood by the engineers, and we found ourselves discussing the avionics and arguing about them over the diagrams, not the statocols... I gradually stopped using the text, or used it only to capture supplementary information inside the states or along transitions, and the diagrams became the actual specification we were constructing...

This was how the basics of the language emerged. I chose to use the term statecharts for the resulting creatures, which was as of 1983 the only unused combination of "state" or "flow" with "chart" or "diagram".

1.5 Short Story of QP

While statecharts solve the first problem, that is the need for "a complete and precise specification of system behaviors (model)", we still need to tackle the second problem, that is the need for "a software architecture that supports the execution of the model".

Quantum Platform is one of such tool that works particular well for embedded systems since it's very light-weight and efficient. This is a short story of its evolution.

1. Miro Samek published his article "State-Oriented Programming" in Embedded System Programming in August 2000.

<http://www.state-machine.com/doc/Samek0008.pdf>
2. In 2002 he published the first edition of his book "Practical Statecharts in C/C++". The 2nd edition came up in 2008.
3. He later formed his company Quantum Leaps with a wide customer base.

QP follows the dual-licensing model (GPL and commercial), with GPL exceptions granted to Raspberry Pi/Arduino/mbed-enabled boards. Check website for latest licensing terms.

Initially QP was called QF (Quantum Framework) and only provides an event processor and event framework. Later he added a kernel called QK and just recently extended its features in QXK. The package as a whole is called QP.

QP is a very active project and is constantly updated to fix bugs and add support to new hardware platforms such as the Cortex-M7. It has ports to many RTOSes including uCOS-II and FreeRTOS, if you choose not to use the bundled QK/QXK.

2 Eclipse/GCC Toolchain

Eclipse with GNU/GCC toolchain is a popular alternative to commercial tools such as the IAR Embedded Workbench. It has the advantage of being free but is more complicated and time-consuming to setup. There are many different flavors. You just need to find one that works for you and stick with it.

The one I use follows a roll-your-own approach with references made to the following online resources:

1. <http://richrobo.blogspot.com/2013/05/opensource-arm-development-using.html>
2. <http://embeddedprogrammer.blogspot.com/2012/09/stm32f4discovery-development-with-gcc.html>
3. <https://www.carminenoviello.com>

The advantage is that you understand what's going on under the hood and can update individual components (e.g. openocd, gcc) when needed. Since it is generic it will work with other Cortex-M processors with some adaptation.

With STMicro acquiring Atollic, the Eclipse/GCC based TrueStudio is now free for STM32 processors. Check out their latest release as the time of writing:

<https://atollic.com/release-news/atollic-truestudio-stm32-9-0-0-released/>

2.1 Quick Setup Procedure

A working Eclipse/GCC development environment has been fully set up in a VirtualBox virtual machine. You can simply launch it on your PC/Mac by following the steps below.

1. Install VirtualBox from the download link on <https://www.virtualbox.org>.
2. Create a folder for your virtual machine on your PC/Mac.

For example, it can be "~/VirtualBox VMs/UbuntuVm" on a Mac.

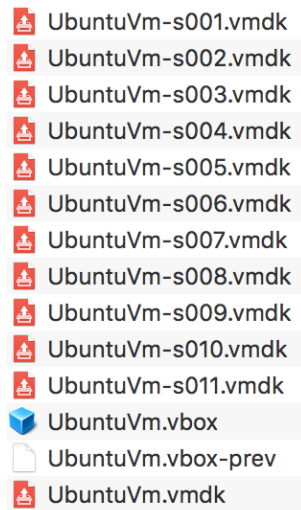
On a PC, it can be "C:\Users\<username>\VirtualBox VMs\UbuntuVm".

3. Download the virtual machine files from

<https://canvas.uw.edu/courses/1188665/files/folder/VirtualBox/UbuntuVm>

Place all downloaded files to the virtual machine folder created in the step above.

The list of files should be similar to this:

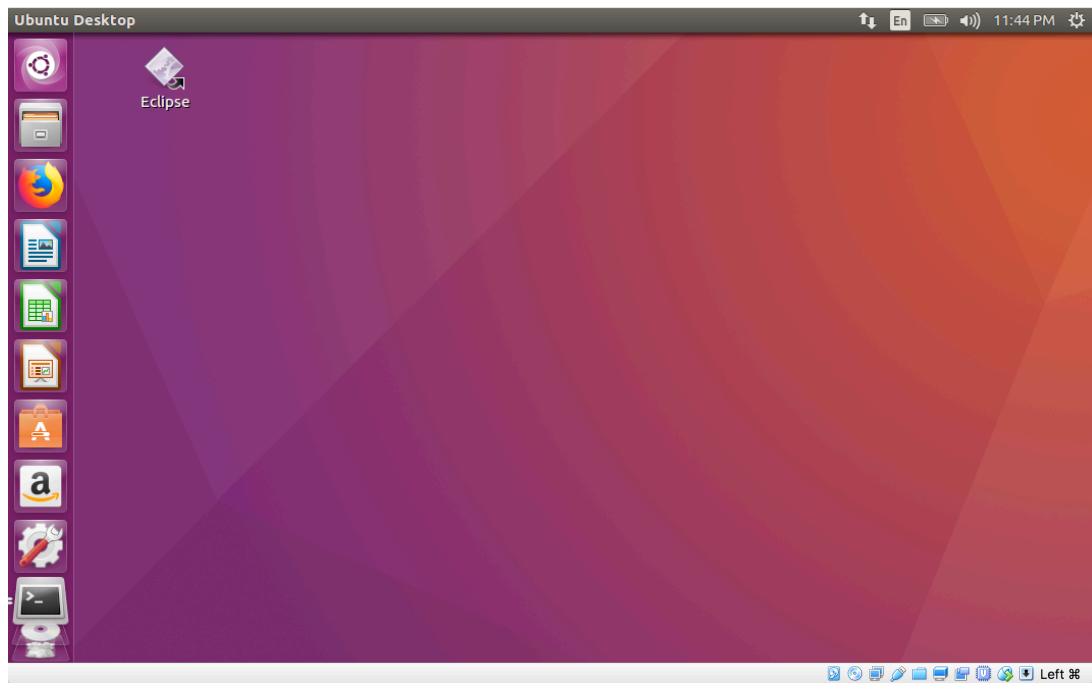


Note: You can download all files in the folder *VirtualBox/UbuntuVm* together as a zip file by clicking the *gear icon* in the row of the *UbuntuVm* folder and selecting *Download*. The downloaded file will be named *VirtualBo_export.zip* and will be extracted to a folder named *UbuntuVm* on your PC. The extracted folder will contain all the .vmdk and .vbox files listed above.

4. The *.vmdk files are for the virtual hard drive. They are divided into multiple files with each less than 2GB for the ease of downloading.
5. Double click **UbuntuVm.vbox** (NOT .vmdk) to launch the virtual machine. You may also open this file from within VirtualBox.

Note – *Do not* create a new virtual machine and mount the UbuntuVm.vmdk files as the hard drive, or you will run into duplicated UUID issues. If you do so you need to run "VBoxManager.exe internalcommands sethduuid <path to UbuntuVm.vmdk>" first.

6. You may need to adjust the settings (e.g. number of processors, memory size, etc.) if you run into errors or warnings when launching the virtual machine.
7. The default Ubuntu username/password is "user/user".
8. If the virtual machine is launched successfully you will see the Ubuntu desktop.



Copyright (C) Lawrence Lo. All rights reserved.

2.2 Detailed Setup Procedure

2.2.1 Virtual Machine

9. Install VirtualBox.
10. Create Ubuntu 16.04 virtual machine with install image ubuntu-16.04.3-desktop-amd64.iso.
Username and password are both set to "user".
11. Install VirtualBox Guest Additions to Ubuntu.

2.2.2 OpenOCD

Build and install openocd from source. This is often required to get support for the latest hardware.

1. Install packages required for building:

```
sudo apt-get install git
sudo apt-get install libusb-1.0
sudo apt-get install libftdi-dev
sudo apt-get install automake
sudo apt-get install libtool-bin
```

2. Clone source:

```
mkdir ~/Projects
cd ~/Projects
git clone http://openocd.zylin.com/p/openocd.git openocd
cd openocd
git submodule init
git submodule update
```

Note – The latest release (0.10) or git master does not yet support STM32H743. Goto gerrit at <http://openocd.zylin.com> to download the checkin labeled as "flash: Add new stm32h7x driver support" (which should include a related checkin "Add STM32H7 config files").

3. ./bootstrap

```
./configure --enable-ftdi --enable-stlink
make -j4
sudo make install
```

4. openocd executable is installed at /usr/local/bin. Set 's' flag by:

```
sudo chmod u+s /usr/local/bin/openocd
```

Note – openocd config files are located at /usr/local/share/openocd

2.2.3 ARM GCC Toolchain

1. To run 32-bit applications on 64-bit Linux, we need to install:

```
sudo apt-get install gcc-multilib
```

In addition GDB requires the following 32-bit library:

```
sudo apt-get install libncurses5:i386
```

2. Download from:

<https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads>

The latest version gcc-arm-none-eabi-6-2017-q2-update gives errors ("reinterpret_cast from integer to pointer") when building the STM32F4 Cube library. The latest version tested to work with the STM32F4 Cube library is gcc-arm-none-eabi-5_4-2016q3.

3. Expand to a folder and move to /usr/local/ as

```
/usr/local/gcc-arm-none-eabi-5_4-2016q3
```

Change permission. Create link by typing

```
ln -s /usr/local/gcc-arm-none-eabi-5_4-2016q3 /usr/local/gcc-arm-none-eabi
```

4. Add path to the end of ~/.bashrc:

```
PATH=$PATH:/usr/local/gcc-arm-none-eabi/bin
```

2.2.4 Eclipse IDE (Neon)

1. Since Eclipse runs on Java, install Java runtime with:

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

2. Download and install "Eclipse IDE for C/C++ Developers" from:

<https://www.eclipse.org/downloads/packages/release/neon/3>

The file to download is named:

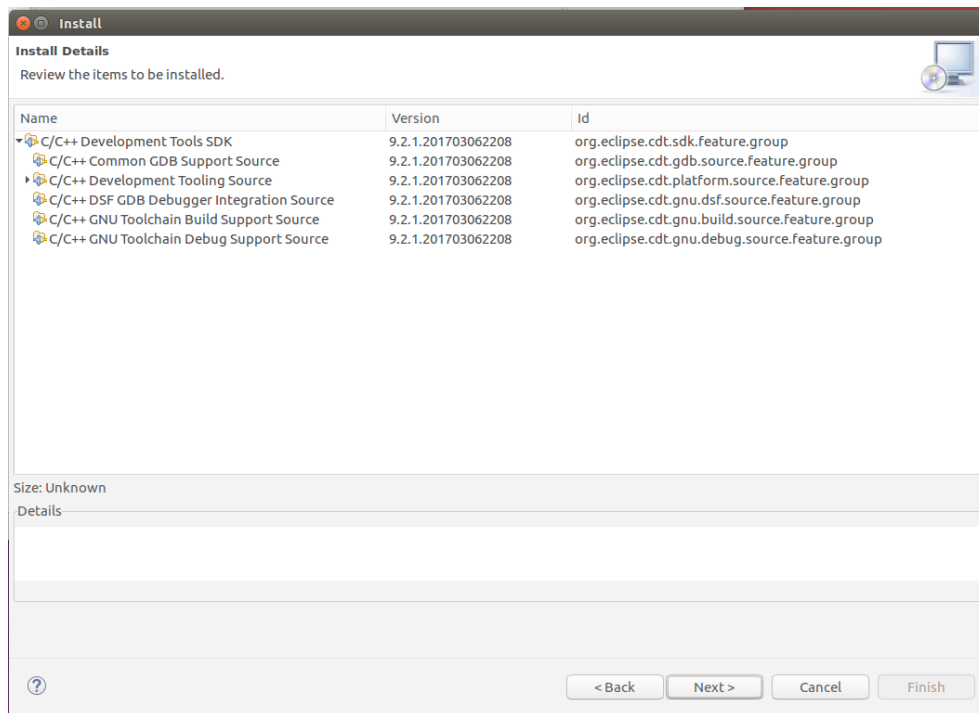
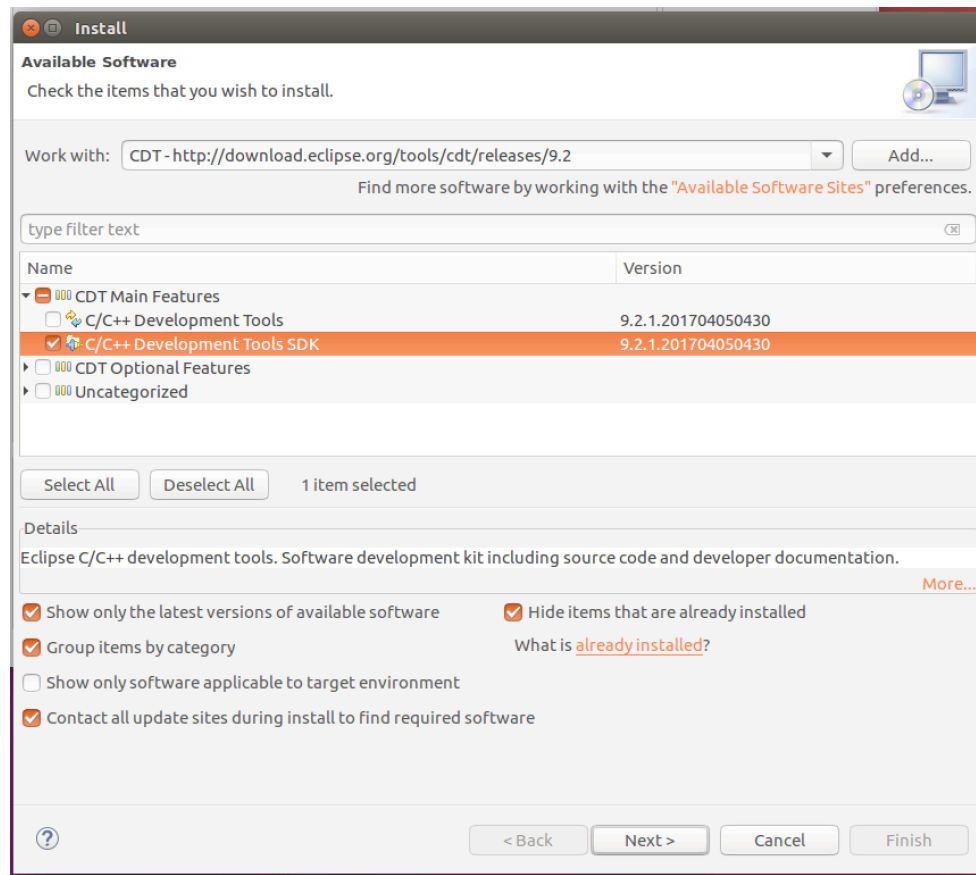
```
eclipse-cpp-neon-3-linux-gtk-x86_64.tar.gz
```

Expand the file under /home/user. The binary to launch is /home/user/eclipse/eclipse.

Make a link to the binary on the Desktop by:

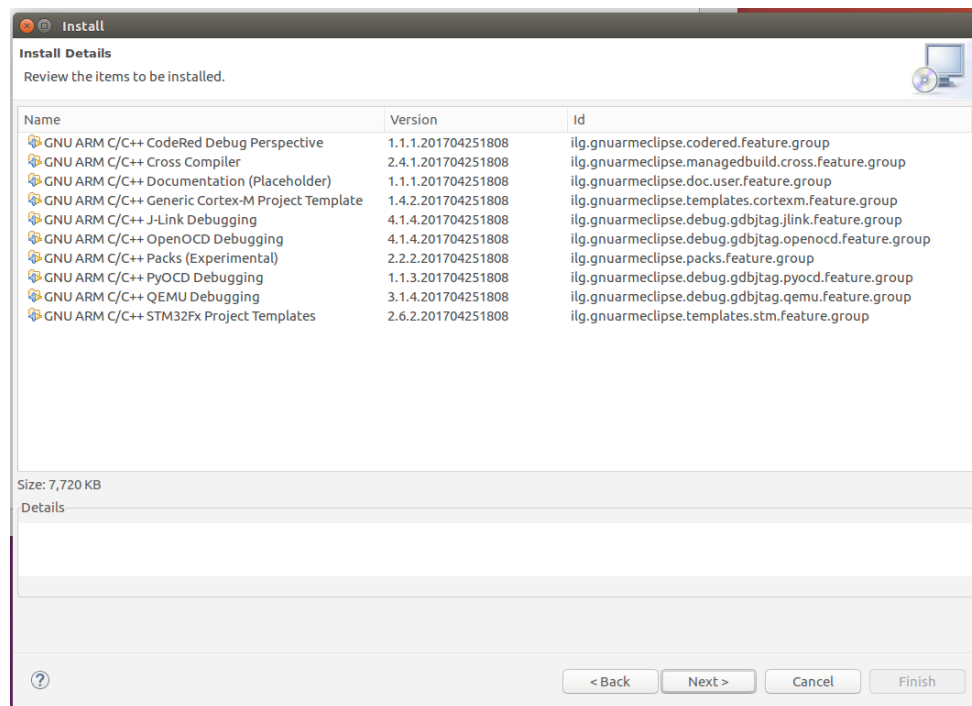
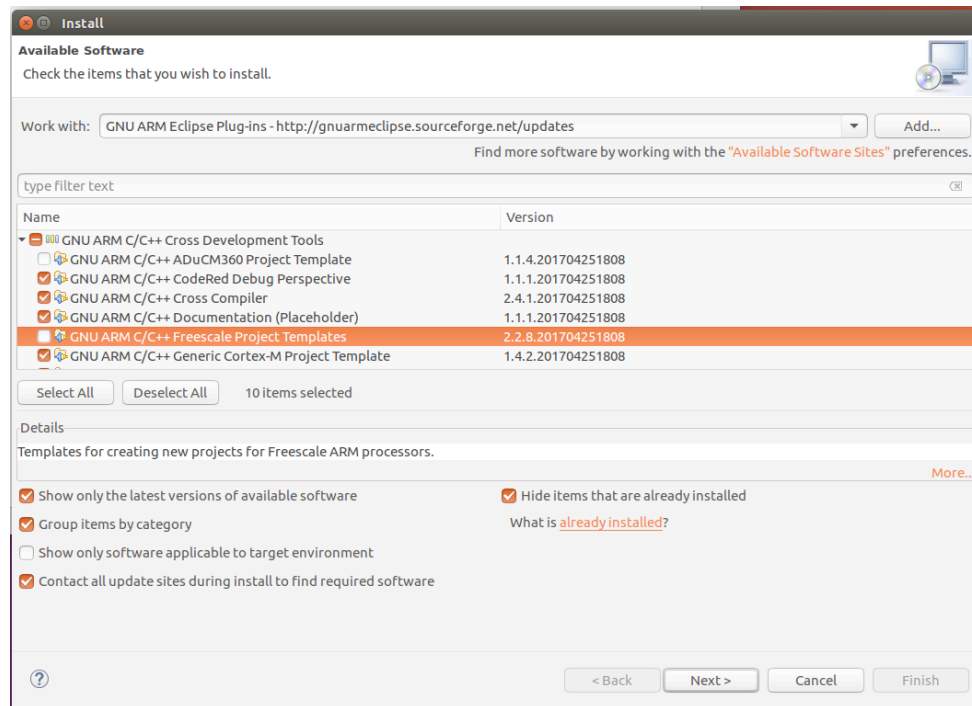
```
ln -s ~/eclipse/eclipse ~/Desktop/Eclipse
```

3. Install "**C/C++ Development Tools SDK**". Launch Eclipse and select "Help -> Install New Software". When prompted about compatibility issues, select the default option to keep current Eclipse installation but modify new software configuration to be compatible.



4. Install GNU ARM Eclipse Plug-ins - <http://gnuarmclipse.sourceforge.net/updates>

Fill in the above text in the textbox "Work with". Expand and select "GNU ARM C/C++ Cross Development Tools". Unselect ADuCM360 and Freescale Project Templates. When prompted about unsigned content warnings, click OK at your own risk.



2.2.5 Eclipse Workspace

1. Download the latest workspace tar file from this link:

<TBD>

The filename should be in the format of "workspace_yyyymmdd.tgz". Place this tar file under ~.

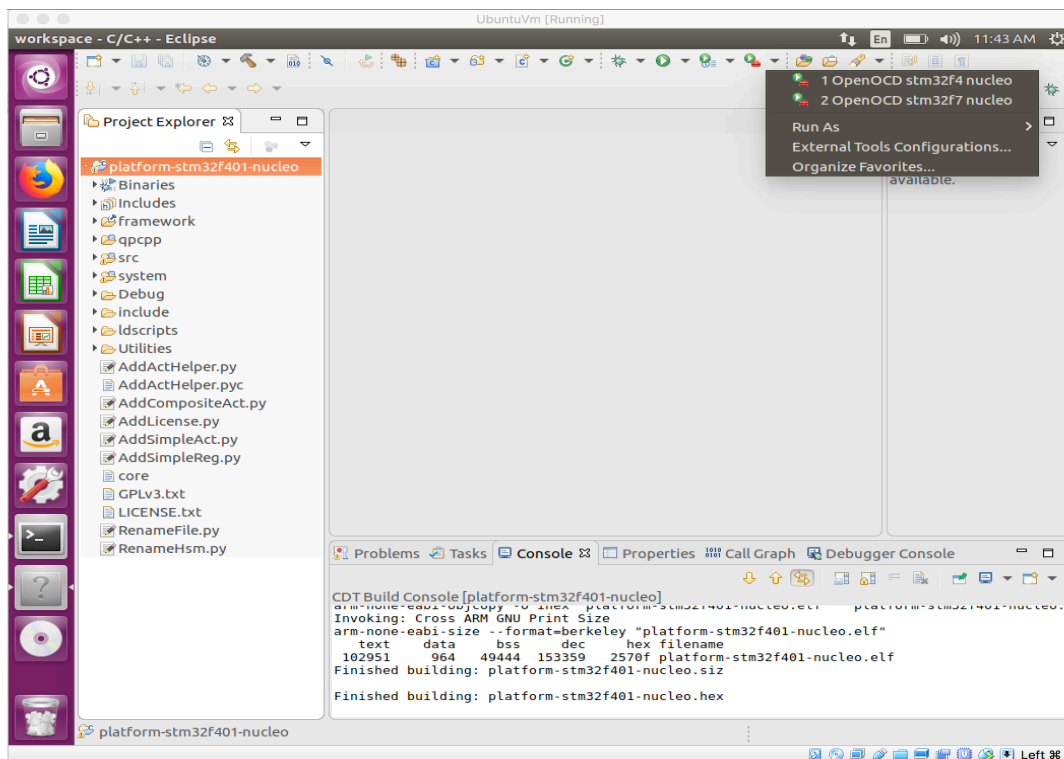
2. When Eclipse is first launched it automatically creates a default workspace under ~/workspace. Backup this default workspace as ~/workspace.bak just in case.
3. Expand the workspace tar file as ~/workspace.
4. The workspace has been preconfigured to support OpenOCD debugging on the STM32F401 Nucleo board. It expects a project for the STM32F401 Nucleo board under ~/Projects/stm32.

Download the latest project tar file from this link:

<TBD>

The filename should be in the format of "platform-stm32f401-nucleo_yyyymmdd.tgz". Create the folder ~/Projects/stm32 and place this tar file under it.

5. Expand the project tar file as ~/Projects/stm32/platform-stm32f401-nucleo.
6. Launch Eclipse and select the default workspace "~/workspace". Double click the project "platform-stm32f401-nucleo" on Project Explorer to see the project folders and files. If the setup is correct you should see something like the following:



3 Build and Debug with Eclipse

3.1 Minicom

You will need a terminal program to communicate with the target board. One such program on Linux is minicom.

1. Install minicom by

```
sudo apt-get install minicom
```

2. Set default configuration by

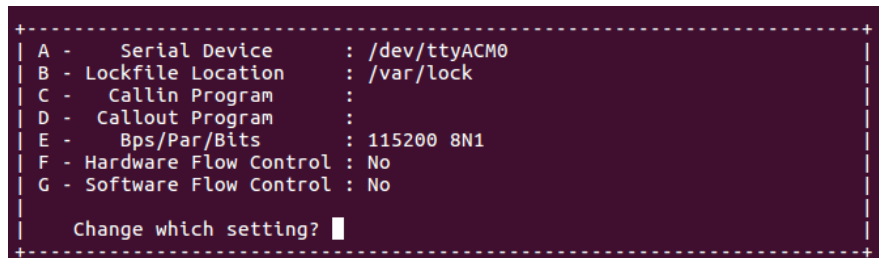
```
sudo minicom -s
```

3. Select "Serial port setup".

Set "Serial Device" to `/dev/ttyACM0`.

Set "Bps/Par/Bits" to `115200 8N1`.

Set "Hardware Flow Control" to `No`.

A screenshot of the minicom serial port setup menu. The menu is displayed in a terminal window with a dark background and light-colored text. It lists several settings: A - Serial Device : /dev/ttyACM0, B - Lockfile Location : /var/lock, C - Callin Program :, D - Callout Program :, E - Bps/Par/Bits : 115200 8N1, F - Hardware Flow Control : No, and G - Software Flow Control : No. At the bottom, it asks "Change which setting?" with a cursor pointing to the first option.

```
+-----+
| A -   Serial Device       : /dev/ttyACM0
| B - Lockfile Location    : /var/lock
| C -   Callin Program      :
| D -   Callout Program     :
| E -   Bps/Par/Bits        : 115200 8N1
| F - Hardware Flow Control : No
| G - Software Flow Control : No
|
| Change which setting? 
+-----+
```

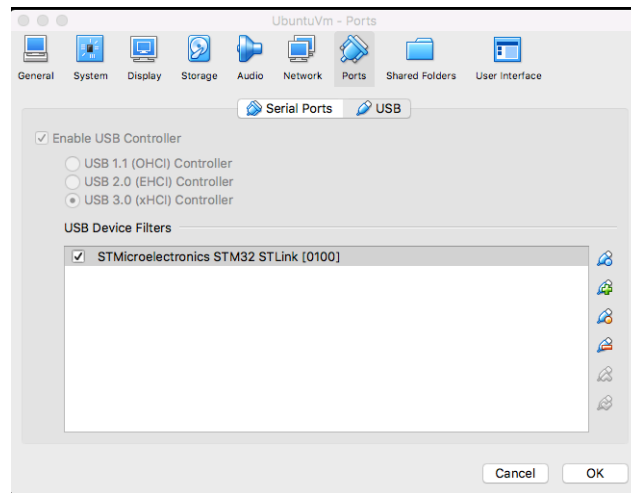
4. When done, hit Enter and select "Save setup as dfl".

3.2 Connect Target.

1. Connect STM32 Nucleo or Discovery board to PC or Mac. Initially the STLink device may be captured by the host machine (PC or Mac).
2. Make sure the USB device "STM32 STLink" is automatically captured by the virtual machine. This is particularly important when using VirtualBox on Mac; otherwise STLink cannot be captured manually.

Click VirtualBox top menu "Devices -> USB -> USB Settings...".

Click the "USB" tab and click the "add device" icon on the right side. Select "STMicroelectronics STM32 STLink". Ensure the checkbox next to the device name is checked.



3.3 Build and Debug

3.3.1 Build Project

1. To build the project, right-click on the project name in Project Explorer. Select "Build Project" to start building the project. You may first select "Clean Project" to get a clean build.
2. There may be warnings but there should be no errors. The output on the Console window at the bottom should look like this if the build is successful:

```

Invoking: Cross ARM GNU Print Size
arm-none-eabi-size --format=berkeley "platform-stm32f401-nucleo.elf"
Invoking: Cross ARM GNU Create Flash Image
arm-none-eabi-objcopy -O ihex "platform-stm32f401-nucleo.elf" "platform-stm32f401-nucleo.hex"
   text  data  bss   dec   hex filename
 102951   964 49444 153359 2570f platform-stm32f401-nucleo.elf
Finished building: platform-stm32f401-nucleo.siz

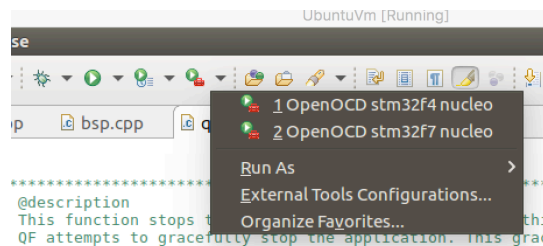
Finished building: platform-stm32f401-nucleo.hex

08:49:43 Build Finished (took 15s.800ms)

```

3.3.2 Launch OpenOCD

1. Ensure the target board is connected to the PC/Mac and the STLink USB device has been captured by the virtual machine.
2. Click on the "Tools" icon on the top toolbar. Select "OpenOCD stm32f4 nucleo".



3. If OpenOCD is launched successfully you should see the following in Console:

```
Open On-Chip Debugger 0.10.0+dev-00207-g4109263-dirty (2018-01-14-15:50)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ
compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v28 API v2 SWIM v18 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.243713
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

4. If OpenOCD fails to launch, try the following:

a) Unplug and replug in the target board. Ensure the STLink USB device is captured by the virtual machine.

b) Terminate any stale OpenOCD process by

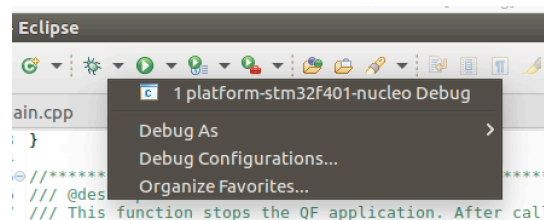
```
ps -A |grep "openocd"
```

```
kill <process id>
```

5. **Note when you unplug the target board while Eclipse is still running, OpenOCD will lose its connection to the target and will show an error message repeatedly in the Console. It will not be able to reconnect when the target is replugged in until you manually terminate the "openocd" process and relaunch it as described above.**

3.3.3 Launch Debug Session

1. Ensure OpenOCD has been successfully launched and connected to the target.
2. Click on the "Debug" icon on the top toolbar. Select "platform-stm32f401-nucleo Debug".



3. If the debug session is launched successfully, you will see the following in the Console:

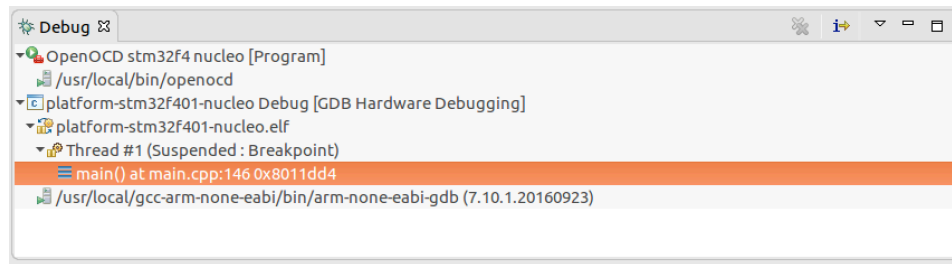
```
...
Info : Padding image section 0 with 8 bytes
Info : Padding image section 1 with 1 bytes
```

```

target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000046 msp: 0x20018000
Warn : keep_alive() was not invoked in the 1000ms timelimit. GDB alive packet not sent! (1097).
Workaround: increase "set remotetimeout" in GDB
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080002bc msp: 0x20018000

```

4. The Debug window should show the program is stopped at the beginning of main():



5. Run the program by clicking the Play/Resume button at the top toolbar:



6. Run minicom in a Ubuntu terminal window:

```
sudo minicom
```

You should see the debug output of the program running on the target which looks like the following (exact output depends on the actual program loaded):

```

...
36 COMPOSITE_ACT(31): Starting COMPOSITE_REG_START_CFM from COMPOSITE_REG0(32) seq=0
36 COMPOSITE_ACT(31): Starting COMPOSITE_REG_START_CFM from COMPOSITE_REG1(33) seq=1
36 COMPOSITE_ACT(31): Starting COMPOSITE_REG_START_CFM from COMPOSITE_REG2(34) seq=2
36 COMPOSITE_ACT(31): Starting COMPOSITE_REG_START_CFM from COMPOSITE_REG3(35) seq=3
36 COMPOSITE_ACT(31): Starting DONE from UNDEF(0) seq=0
36 SYSTEM(1): Root COMPOSITE_ACT_START_CFM from COMPOSITE_ACT(31) seq=0
36 COMPOSITE_ACT(31): Starting EXIT
37 COMPOSITE_ACT(31): Started ENTRY

```

4 Q&A