AGL/Phase2 - Devkit

Build from scratch: AGL image and SDK for Porter

Version 2.0

July 2016

Abstract

The AGL DevKit allows developers to rebuild a complete bootable image and its associated SDK from source code. It uses Yocto/Poky version 2.x with latest version of Renesas BSP and enables low-level development of drivers and system services.

The AGL DevKit contains:

- This guide, which describes how to create a Docker container able to build AGL images and associated SDKs. The container is also suitable to build AGL Applications independently of Yocto/Bitbake.
- Applications templates and demos available on Github, to start developing various types of applications independently of Yocto:
  - services
  - native applications
  - HTML5 applications
  - ...
- A documentation guide "**AGL Devkit – Build your 1st AGL Application**" which explains how to use the AGL SDK to create applications based on templates.

This document focuses on building from scratch an AGL image for Porter board, within a Docker container, and then install the associated SDK.

Document revisions

Date	Version	Designation	Author
11 Feb. 2016	0.1	Initial release	S. Desneux [Iot.bzh] Y. Gicquel [Iot.bzh]
18 Feb. 2016	0.2	Fixes, multi-platform (Windows, Mac)	M.Bachmann [Iot.bzh]
29 Feb. 2016	0.3	Fixes for multi-platform	M.Bachmann [Iot.bzh]
29 Feb. 2016	1.0	Final Review	S. Desneux [IoT.bzh]
29 Mar. 2016	1.1	Public version without proprietary drivers	S. Desneux [IoT.bzh] Y. Gicquel [Iot.bzh]
15 Jul. 2016	2.0	Update for AGL 2.0, SDK generation & installation added	S. Desneux [IoT.bzh] M.Bachmann [IoT.bzh]

Table of contents

1. Deploy an image using containers

1.1. Motivation

The Yocto build environment is subject to many variations depending on:

- Yocto/Poky/OpenEmbedded versions and revisions
- Specific layers required for building either the BSP or the whole distribution
- Host distribution and version[1]
- User environment

In particular, some recent Linux host distributions (Ubuntu 15.x, Debian 8.x, OpenSUSE 42.x, CentOS 7.x) do not officially support building with Yocto 2.0. Unfortunately, there's no easy solution to solve this kind of problem: we will still observe for quite a long time a significant gap between the latest OS versions and a fully certified build environment.

To circumvent those drawbacks and get more deterministic results amongst the AGL community of developers and integrators, using virtualization is a good workaround. A Docker container is now available for AGL images: it is faster, easier and less error-prone to use a prepared Docker container because it provides all necessary components to build and deploy an AGL image, including a validated base OS, independently of the user's host OS. Moreover, light virtualization mechanisms used by Docker do not add much overhead when building: performances are nearly equal to native mode.

1.2. Prerequisites

To run an AGL Docker image, the following prerequisites must be fulfilled:

- You must run a 64-bit operating system, with administrative rights,
- Docker engine v1.8 or greater must be installed,
- An internet connection must be available to download the Docker image on your local host.

1 The list of validated host distros is defined in the Poky distro, in the file meta-yocto/conf/distro/poky.conf and also at http://www.yoctoproject.org/docs/2.0/ref-manual/ref-manual.html#detailed-supported-distros

2. Setting up your operating system

In this section, we describe the Docker installation procedure depending on your host system. We will be focusing on the most popular systems; for a full list of supported operating systems, please refer to Docker online documentation: http://docs.master.dockerproject.org/

2.1. Linux (Ubuntu / Debian)

At the time of writing, Docker project supports these Ubuntu/Debian releases:

- Ubuntu Xenial 16.04 LTS
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 LTS
- Ubuntu Precise 12.04 LTS
- Debian 8.0 (64-bit)
- Debian 7.7 (64-bit)

For an updated list of supported distributions, you can refer to the Docker project website, at these locations:

- http://docs.master.dockerproject.org/installation/debian/
- http://docs.master.dockerproject.org/installation/ubuntulinux/

Here are the commands needed to install the Docker engine on your local host:

```
sudo apt-get update
sudo apt-get install wget curl
wget -qO- https://get.docker.com/ | sh
```

This will register a new location in your "sources.list" file and install the "docker.io" package and its dependencies:

```
cat /etc/apt/sources.list.d/docker.list
deb https://apt.dockerproject.org/repo ubuntu-trusty main
docker --version
Docker version 1.9.1, build a34a1d5
```

It is then recommended to add your user to the new "docker" system group:

```
sudo usermod -aG docker <your-login>
```

… and after that, to log out and log in again to have these credentials applied.

You can reboot your system or start the Docker daemon using:

```
sudo service docker start
```

If everything went right, you should be able to list all locally available images using:

```
docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
```

In our case, no image is available yet, but the environment is ready to go on.

A SSH client must also be installed:

```
sudo apt-get install openssh-client
```

2.2. Windows© (7, 8, 8.1, 10)

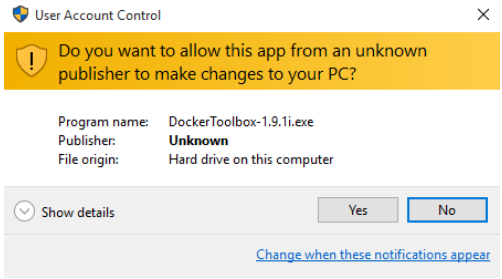
WARNING: although Windows© can work for this purpose, not only are lots of additional steps needed, but the build process performance itself is suboptimal. Please consider moving to Linux for a better experience.

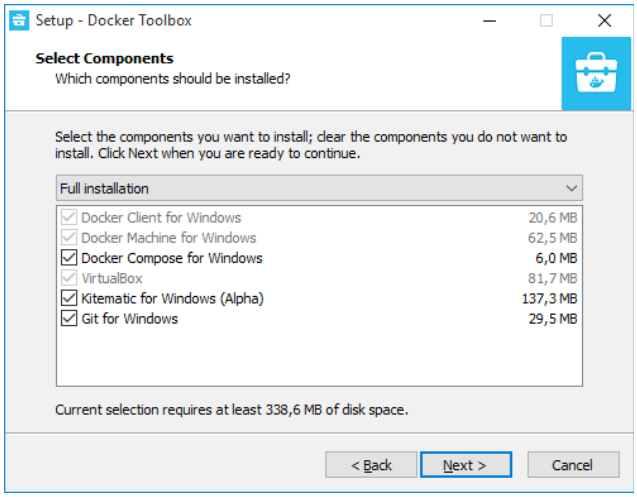
We will be downloading the latest Docker Toolbox at the following location:

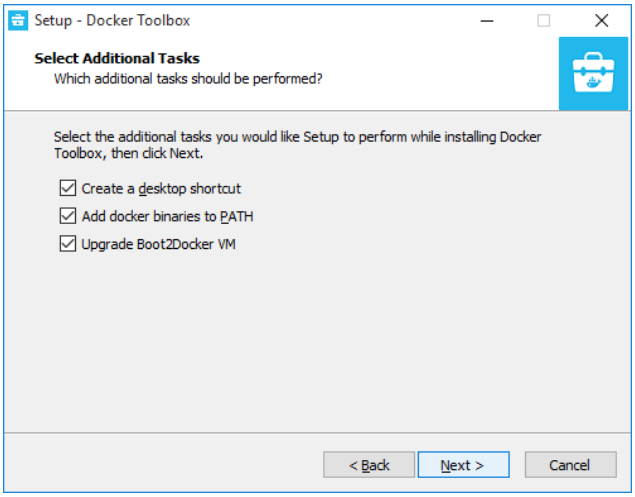
https://www.docker.com/docker-toolbox

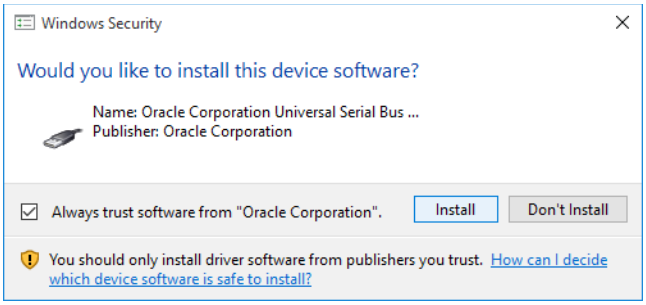
and by clicking on the "*Download (Windows)*" button:

We will answer "Yes", "Next" and "Install" in the next dialog boxes.

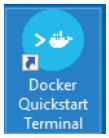




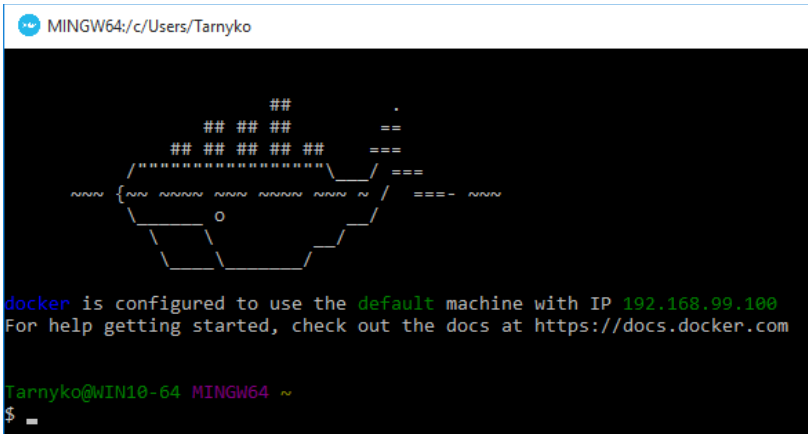




We can then start it by double-clicking on the "Docker Quickstart Terminal" icon:



It will take a certain amount time to setup everything, until this banner appears:



Docker Toolbox provides a 1 Gb RAM/20 Go HDD virtual machine; this is clearly insufficient for our needs. Let us expand it to 4 Gb RAM/50 HDD[2]:

```
export VBOXPATH=/c/Program\ Files/Oracle/VirtualBox/
export PATH="$PATH:$VBOXPATH"

docker-machine stop default

VBoxManage.exe modifyvm default --memory 4096
VBoxManage.exe createhd --filename build.vmdk --size 51200 --format VMDK
VBoxManage.exe storageattach default --storagectl SATA --port 2 \
  --medium build.vmdk --type hdd

docker-machine start default

docker-machine ssh default "sudo /usr/local/sbin/parted --script /dev/sdb \
  mklabel msdos"
docker-machine ssh default "sudo /usr/local/sbin/parted --script /dev/sdb \
  mkpart primary ext4 1% 99%"
docker-machine ssh default "sudo mkfs.ext4 /dev/sdb1"
docker-machine ssh default "sudo mkdir /tmp/sdb1"
docker-machine ssh default "sudo mount /dev/sdb1 /tmp/sdb1"
docker-machine ssh default "sudo cp -ra /mnt/sda1/* /tmp/sdb1"

docker-machine stop default

VboxManage.exe storageattach default --storagectl SATA --port 2 --medium none
VboxManage.exe storageattach default --storagectl SATA --port 1 \
  --medium build.vmdk --type hdd

docker-machine start default
```

2 These are minimal values; feel free to increase them if your computer has more physical memory and/or free space.

We will then finalize the setup:

```
VboxManage.exe modifyvm default --natpf1 sshredir,tcp,127.0.0.1,2222,,2222
docker-machine start default
docker-machine ssh default "echo mkdir /sys/fs/cgroup/systemd | \
  sudo tee /var/lib/boot2docker/bootlocal.sh"
docker-machine restart default
```

A SSH client must also be installed. We will grab *PuTTY* at the following URL:

http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe

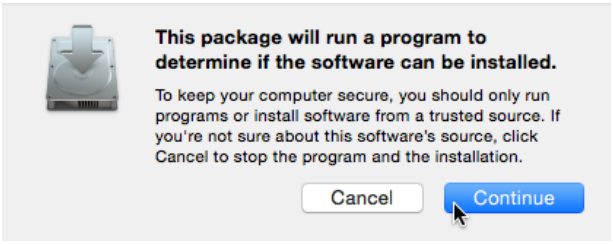
2.3. Mac OS X©

We will be downloading the latest Docker Toolbox at the following location:

https://www.docker.com/docker-toolbox

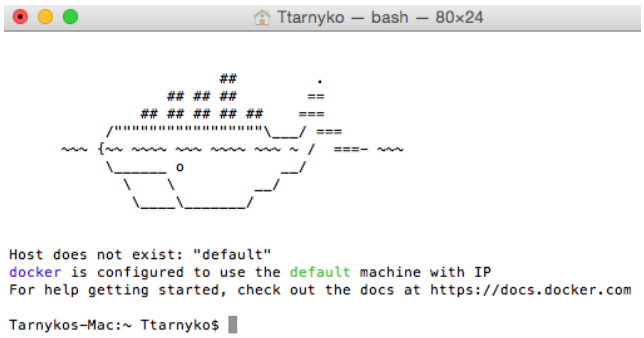
and by clicking on the "*Download (Mac)*" button:

We will answer "Continue" and "Install" in the next dialog boxes:



Then, when we go to our "Applications" folder, we now have a "Docker" subfolder where we can start "Docker Quickstart Terminal":

It will take a certain amount of time to setup everything, until this banner appears:



Docker Toolbox provides a 1 Gb RAM/20 Go HDD virtual machine; this is clearly insufficient for our needs. Let us expand it to 4 Gb RAM/50 HDD[3]:

```
docker-machine stop default

VboxManage modifyvm default --memory 4096
VboxManage createhd --filename build.vmdk --size 51200 --format VMDK
VboxManage storageattach default --storagectl SATA --port 2 \
  --medium build.vmdk --type hdd

docker-machine start default

docker-machine ssh default "sudo /usr/local/sbin/parted --script /dev/sdb \
  mklabel msdos"
docker-machine ssh default "sudo /usr/local/sbin/parted --script /dev/sdb \
  mkpart primary ext4 1% 99%"
docker-machine ssh default "sudo mkfs.ext4 /dev/sdb1"
docker-machine ssh default "sudo mkdir /tmp/sdb1"
docker-machine ssh default "sudo mount /dev/sdb1 /tmp/sdb1"
docker-machine ssh default "sudo cp -ra /mnt/sda1/* /tmp/sdb1"

docker-machine stop default

VboxManage storageattach default --storagectl SATA --port 2 --medium none
VboxManage storageattach default --storagectl SATA --port 1 \
  --medium build.vmdk --type hdd

docker-machine start default
```

3 These are minimal values; feel free to increase them if your computer has more physical memory and/or free space.

We will then finalize the setup:

```
VboxManage modifyvm default --natpf1 sshredir,tcp,127.0.0.1,2222,,2222
docker-machine ssh default "echo mkdir /sys/fs/cgroup/systemd | \
  sudo tee /var/lib/boot2docker/bootlocal.sh"
docker-machine restart default
```

3. Install AGL Yocto image for Porter board using Docker container

3.1. Overview

This section gives details on a procedure which allows system developers and integrators to set up a the build environment image on their local host.

The prepared environment is deployed and available thanks to lightweight virtualization containers using Docker technology[4]. The pre-built image for AGL development activities is currently designed to be accessed using SSH Protocol.

3.2. Download the image from the registry

To download the image, you can enter:

```
docker pull docker.iot.bzh/iotbzh/worker_bsp_base:latest
latest: Pulling from iotbzh/worker_bsp_base
42755cf4ee95: Downloading 11.09 MB/130.9 MB
5f70bf18a086: Download complete
b15a7afd6484: Download complete
e3d7ee351c85: Downloading 12.14 MB/60.69 MB
9e79ad91095e: Verifying Checksum
bb96a5b9fda6: Downloading 10.56 MB/731.1 MB
```

This operation will take some time as around 800 MB of filesystem layers need to be pulled from the remote repository.

Alternatively, we also distribute the image as a compressed archive which can be downloaded faster as its footprint is around 130 MB. You can then import it into Docker with the following command:

```
curl http://iot.bzh/download/public/2016/bsp/docker_iotbzh_worker_bsp_base-latest.tar.xz | docker load
```

Whatever the download method, the new Docker image should be available. This can be checked by running 'docker images':

```
docker images
REPOSITORY                                TAG     IMAGE ID         CREATED          VIRTUAL SIZE
docker.iot.bzh/iotbzh/worker_bsp_base     latest  90dbaa376d07     2 days ago       885.4 MB
```

4 See https://www.docker.com/

3.3. Start the container

Once the image is available on your local host, you can start the container and the SSH service. We'll also need a local directory on the host to store bitbake mirrors (download cache and sstate cache): this mirror helps to speed up builds.

First, create a local directory and make it available to everyone:

```
MIRRORDIR=<your path here, ~/mirror for example>
mkdir -p $MIRRORDIR
chmod 777 $MIRRORDIR
```

Then we can start the docker image using the following command:

```
docker run \
    --publish=2222:22 \
    --publish=8000:8000 \
    --publish=69:69/udp --publish=10809:10809 \
    --detach=true --privileged \
    --hostname=bsp-devkit --name=bsp-devkit \
    -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
    -v $MIRRORDIR:/home/devel/mirror \
    docker.iot.bzh/iotbzh/worker_bsp_base:latest
```

Then, you can check that the image is running with the following command:

```
docker ps
CONTAINER ID        IMAGE                                                COMMAND                  CRE-
ATED             STATUS              PORTS
NAMES
7037f509509c        docker.iot.bzh/iotbzh/worker_bsp_base:latest   "/usr/bin/wait_for_ne"   5
seconds ago       Up 2 seconds        0.0.0.0:2222->22/tcp, 0.0.0.0:69->69/udp, 0.0.0.0:8000-
>8000/tcp, 0.0.0.0:10809->10809/tcp   bsp-devkit
```

The container is now ready to be used. A dedicated user has been declared:

- login: **devel**
- password: **devel**

The following port redirections allow access to some services in the container:

- port 2222: SSH access using 'ssh -p 2222 devel@localhost'
- port 8000: access to Toaster WebUI through http://localhost:8000/ when started (see Yocto documentation)
- ports 69 (TFTP) and 10809 (NBD): used for network boot (future enhancement)

For easier operations, you can copy your ssh identity inside the container:

```
ssh-copy-id -p 2222 devel@localhost  # password is 'devel'
```

3.4. Connect to Yocto container through SSH

The DevKit container provides a pre-built set of tools which can be accessed through a terminal by using Secure Shell protocol (SSH).

3.4.1. Linux, Mac OS X©

On Linux-based systems, you may need to install an SSH client.

To launch the session, you can enter the following under Linux or Mac OS X:

```
ssh -p 2222 devel@localhost
```

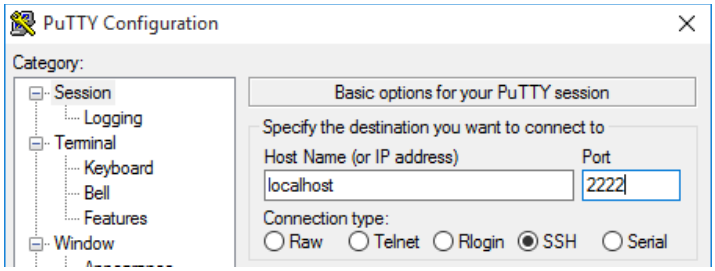
The password is "**devel**". You should obtain the following prompt after success:

```
devel@localhost's password: devel

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
[11:28:27] devel@bsp-devkit:~$
```

3.4.2. Windows©

You will need *PuTTY*, downloaded during the setup section. Run it using its icon:

We can then connect to "l**ocalhost**" on port "**2222**".



Credentials are the same as for Linux: user is "**devel**" with password "**devel**".

3.5. Set up a persistent workspace

AGL Docker image brings a set of tools and here we describe a way to prepare a "shared directory" on your local host accessible from the container. The aim of this shared directory is to allow your ongoing developments to stay independent from the container upgrades.

Please note this whole procedure is not mandatory, but highly recommended as it will save disk space later when you will deploy the SD card image on your target.

3.5.1. From Linux host using a shared directory

Current docker implementation has a limitation about UID:GID mapping between hosts and containers. In the long run, the planned mechanism is to use the "user namespace" feature. But for now, we propose another approach unfortunately less flexible.

We can use a directory on the local host with a dedicated Unix group using a common GID between the host and the container. This GID has been fixed to "1664"[5] and can be created on your linux host using the following commands:

```
sudo groupadd --gid 1664 agl-sdk
sudo usermod -aG agl-sdk <your-login>
```

If this GID is already used on your local host, you will have to use it for this sharing purpose as well. In case this is not possible, another option to exchange workspace data can be the use of a network service (like SSH, FTP) of the container and from your local host.

Once the GID is ready to use, we can create a shared directory (not as 'root', but as your normal user):

```
cd
mkdir $HOME/agl-workspace
sudo chgrp agl-sdk $HOME/agl-workspace
chmod ug+w $HOME/agl-workspace
```

And run the Docker image with the new highlighted switch:

```
docker run \
    --publish=2222:22 \
    --publish=8000:8000 \
    --publish=69:69/udp --publish=10809:10809 \
    --detach=true --privileged \
    --hostname=bsp-devkit --name=bsp-devkit \
    -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
    -v $MIRRORDIR:/home/devel/mirror \
    -v $HOME/agl-workspace:/xdt/workspace \
     docker.iot.bzh/iotbzh/worker_bsp_base:latest
```

5 https://en.wikipedia.org/wiki/Beer_in_France

3.5.2. From Windows© host using a shared directory

We will create a shared directory for our user:

```
mkdir /c/Users/$USERNAME/agl-workspace
```

And run the Docker image with the new highlighted switch:

```
docker run --publish=2222:22 --publish=8000:8000 \
    --publish=69:69/udp --publish=10809:10809 \
    --detach=true --privileged --hostname=bsp-devkit --name=bsp-devkit \
    -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
    -v $MIRRORDIR:/home/devel/mirror \
    -v /c/Users/$USERNAME/agl-workspace:/xdt/workspace \
    docker.iot.bzh/iotbzh/worker_bsp_base:latest
```

3.5.3. From the container using a remote directory (SSHFS)

It's also possible to mount a remote directory inside the container if the source host is running a ssh server. In that case, we will use a SSH connection from the host to the container as a control link, and another SSH connection from the container to the host as a data link.

To do so, you can start the container normally as described in 3.3, start an SSH session and run the following commands to install the package "sshfs" inside the container:

```
sudo apt-get update
sudo apt-get install -y sshfs
```

NB: sudo will ask for the password of the user "**devel**", which is "**devel**".

Now, if we want to mount the remote dir '/data/workspace' with user 'alice' on host 'computer42', then we would run:

```
sshfs alice@computer42:/data/workspace -o nonempty $XDT_WORKSPACE
…
Password: <enter alice password on computer42>
```

NB: the directory on the remote machine must be owned by the remote user

Verify that the mount is effective:

```
df /xdt/workspace
Filesystem                          1K-blocks    Used Available Use% Mounted on
alice@computer42:/data/workspace 103081248 7138276  95612016   7% /xdt/workspace
```

From now, the files created inside the container in /xdt/workspace are stored 'outside', in the shared directory with proper uid/gid.

To unmount the shared directory, you can run:

```
sudo umount $XDT_WORKSPACE
```

4. Inside the container

4.1. Features

Container features:

- a Debian 8.5 based system with an SSH server listening on tcp/22,
- a dedicated user is defined to run the SSH session: **devel** (password: **devel**)
- a script named "prepare_meta" for preparing the build environment

4.2. File system organization and shared volume

The image has been designed with a dedicated file-system hierarchy. Here it is:

```
devel@bsp-devkit:/$ tree -L 2 /xdt
/xdt
|-- build
|   `-- conf
|       |-- bblayers.conf
|       |-- local.conf
|     [snip]
|-- ccache
|-- downloads
|-- meta
|   |-- agl-manifest
|   |-- meta-agl
|   |-- meta-agl-demo
|   |-- meta-agl-extra
|   |-- meta-amb
|   |-- meta-intel
|   |-- meta-intel-iot-security
|   |-- meta-iot-agl
|   |-- meta-oic
|   |-- meta-openembedded
|   |-- meta-qt5
|   |-- meta-renesas
|   |-- meta-rust
|   |-- meta-security-isafw
|   `-- poky
|-- sdk
|-- sources
|-- sstate-cache
|   |-- 00
|   |-- 01
|   |-- 02
|   |-- 03
|   |-- 04
|   |-- 05
|   |-- 06
|   |-- 07
  [snip]
`-- workspace
```

Noticeably, the BSP related features are located in the dedicated "/xdt" directory.

This directory contains sub-directories, and in particular the following:

- **build**: will contain the result of the build process, including an image for the Porter board.
- **downloads**: (optional) contain the Yocto download cache, a feature which will locally store the upstream projects sources codes and which is fulfilled when an image is built for the first time. When populated, this cache allow the user to built without any connection to Internet.
- **meta**: contains the pre-selected Yocto layers required to built the relevant AGL image for the Porter board.
- **sstate-cache**: (optional) contain the Yocto shared state directory cache, a feature which store the intermediate output of each task for each recipe of an image. This cache enhance the image creation speed time by avoiding Yocto task to be run when they are identical to a previous image creation.

5. Build an image for Porter board

In this section, we will go on the image compilation for the Porter board within the Docker container.

5.1. Download Renesas proprietary drivers

For the Porter board, we first need to download the proprietary drivers from Renesas web site. The evaluation version of these drivers can be found here:

http://www.renesas.com/secret/r_car_download/rcar_demoboard.jsp

under the **Target hardware: R-Car H2, M2 and E2** section:

Target hardware: R-Car H2, M2 and E2



Product name: R-Car Series Evaluation Software Package for Linux

Product Name	Download File
R-Car Series Evaluation Software Package for Linux	To download Multimedia and Graphics library, please click this link. Go to Download page »
	To download related Linux drivers, please click here .

Note that you have to register with a free account on MyRenesas and accept the license conditions before downloading them. The operation is fast and simple but nevertheless mandatory to access evaluation of non open-source drivers for free.

Once you register, you can download two zip files: store them in a directory visible from the container, for example in the directory $MIRRORDIR/proprietary-renesas-r-car ($MIRRORDIR was created previously in section 3.3) and adjust the permissions. The zip files should then be visible from the inside of the container in /home/devel/mirror:

```
$ chmod +r /home/devel/mirror/proprietary-renesas-r-car/*.zip
$ ls -l /home/devel/mirror/proprietary-renesas-r-car/
total 8220
-rw-r--r-- 1 1000 1000 6047383 Jul 11 11:03 R-
Car_Series_Evaluation_Software_Package_for_Linux-20151228.zip
-rw-r--r-- 1 1000 1000 2394750 Jul 11 11:03 R-
Car_Series_Evaluation_Software_Package_of_Linux_Drivers-20151228.zip
```

5.2. Setup the build environment

We should first prepare the environment to build images.

This can be easily done thanks to a helper script named “prepare_meta”. This script does the following:

- check for an updated version at https://github.com/iotbzh/agl-manifest
- pull Yocto layers from git repositories, following a snapshot manifest
- setup build configuration (build/conf files)

The following options are available:

```
devel@bsp-devkit:~$ prepare_meta -h
prepare_meta [options]

Options:
   -f|--flavour <flavour[/tag_or_revision]>
      what flavour to us
      default: 'iotbzh'
      possible values: 'stable','unstable','testing', 'iotbzh' ... see agl-
manifest git repository
   -o|--outputdir <destination_dir>
      output directory where subdirectories will be created: meta, build, ...
      default: current directory (.)
   -l|--localmirror <directory>
      specifies a local mirror directory to initialize meta, download_dir or
sstate-cache
       default: none
   -r|--remotemirror <url>
      specifies a remote mirror directory to be used at build time for down-
load_dir or sstate-cache
       default: none
   -p|--proprietary <directory>
      Directory containing Renesas proprietary drivers for RCar platform (2 zip
files)
       default: none
   -e|--enable <option>
      enable a specific option
       available options: ccache, upgrade
   -d|--disable <option>
      disable a specific option
       available options: ccache, upgrade
   -t|--target <name>
      target platform
       default: porter
       valid options: intel-corei7-64, qemux86, qemux86-64, wandboard
   -h|--help
      get this help

Example:
     prepare_meta -f iotbzh/master -o /tmp/xdt -l /ssd/mirror -p /vol/xdt/pro-
prietary-renesas-rcar/ -t porter
```

In our case, we can start it with the following arguments:

- build in /xdt (-o /xdt)
- build for porter board (-t porter)
- build the 'iotbzh' flavour (-f iotbzh), which contains the standard AGL layers + security and app framework. Flavours are stored in the agl-manifest repository.
- Use a local mirror (-l <mirror path>). This directory may contain some directories generated in a previous build: 'downloads', 'sstate-cache', 'ccache', 'meta'. If found, the mirror directories will be specified in configuration files.
- specify proprietary drivers location (-p <drivers path>)

So we can run the helper script:

```
devel@bsp-devkit:~$ prepare_meta -o /xdt -t porter -f rel2.0 -l
/home/devel/mirror/ -p /home/devel/mirror/proprietary-renesas-r-car/ -e
wipeconfig
[…]
=== setup build for porter
Using proprietary Renesas drivers for target porter
=== conf: build.conf
=== conf: download caches
=== conf: sstate caches
=== conf: local.conf
=== conf: bblayers.conf.inc -> bblayers.conf
=== conf: porter_bblayers.conf.inc -> bblayers.conf
=== conf: bblayers_proprietary.conf.inc is empty
=== conf: porter_bblayers_proprietary.conf.inc is empty
=== conf: local.conf.inc is empty
=== conf: porter_local.conf.inc is empty
=== conf: local_proprietary.conf.inc is empty
=== conf: porter_local_proprietary.conf.inc is empty
==========================================================================

Build environment is ready. To use it, run:

# source /xdt/meta/poky/oe-init-build-env /xdt/build

then

# bitbake agl-demo-platform
```

Now, the container shell is ready to build an image for Porter.

5.3. Launch the build

To start the build, we can simply enter the indicated commands:

```
devel@bsp-devkit:~$ . /xdt/build/agl-init-build-env
### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common target are:
    agl-demo-platform
devel@bsp-devkit:/xdt/build$ bitbake agl-demo-platform
[snip]
NOTE: Tasks Summary: Attempted 5108 tasks of which 4656 didn't need to be rerun
and all succeeded.

Summary: There were 19 WARNING messages shown.
devel@bsp-devkit:/xdt/build$
```

Without mirror, it will take a few hours to build all the required component of the AGL distribution, depending on: your host machine CPU, disk drives types and internet connection.

5.4. Updating the local mirror

Optionally, at the end of the build, some directories may be synced to the mirror dir, for future usage:

- /xdt/meta: contains all layers used to build AGL
- /xdt/downloads: download cache (avoid fetching source from remote sites)
- /xdt/sstate-cache: binary build results (avoid recompiling sources)

This can be done with the following command:

```
devel@bsp-devkit:~$ for x in meta downloads sstate-cache; do rsync -Pav \
--delete /xdt/$x /home/devel/mirror/$x; done
```

6. Porter image deployment on target

Once the Porter image has been built with Yocto, we can deploy it on an empty SD card to prepare its use on the target.

6.1. SD card image creation

First, we need to generate an SD card disk image file. For this purpose, a helper script is provided within the container. Here below is the way to use it.

6.1.1. Linux, Mac OS X©

```
devel@bsp-devkit:/xdt/build$ $ mksdcard /xdt/build/tmp/deploy/images/porter/agl-demo-platform-porter-20XXYYZZxxyyzz.rootfs.tar.bz2 /home/devel/mirror
```

6.1.2. Windows©

```
devel@bsp-devkit:/xdt/build$ sudo dd if=/dev/zero of=/sprs.img bs=1 count=1 seek=4G
devel@bsp-devkit:/xdt/build$ sudo mkfs.ext4 /sprs.img
devel@bsp-devkit:/xdt/build$ sudo mkdir /tmp/sprs
devel@bsp-devkit:/xdt/build$ sudo mount /sprs.img /tmp/sprs
devel@bsp-devkit:/xdt/build$ sudo mksdcard /xdt/build/tmp/deploy/images/porter/agl-demo-platform-porter-20XXYYZZxxyyzz.rootfs.tar.bz2 /tmp/sprs/sdcard.img
devel@bsp-devkit:/xdt/build$ xz -dc /tmp/sprs/sdcard.img.xz > $XDT_WORKSPACE/agl-demo-platform-porter-sdcard.img
```

You should get the following prompt during the "mksdcard" step:

```
Creating the image agl-demo-platform-porter-sdcard.img ...
0+0 records in
0+0 records out
0 bytes (0 B) copied, 6.9187e-05 s, 0.0 kB/s
mke2fs 1.42.12 (29-Aug-2014)
Discarding device blocks: done
Creating filesystem with 524287 4k blocks and 131072 inodes
Filesystem UUID: 5307e815-9acd-480b-90fb-b246dcfb28d8
Superblock backups stored on blocks:
      32768, 98304, 163840, 229376, 294912

Allocating group tables: done
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

Extracting image tarball...
done


Image agl-demo-platform-porter-sdcard.img created!

Set the following uboot environment
setenv bootargs_console 'console=ttySC6,38400 ignore_loglevel'
setenv bootargs_video   'vmalloc=384M video=HDMI-A-1:1920x1080-32@60'
setenv bootargs_root    'root=/dev/mmcblk0p1 rootdelay=3 rw rootfstype=ext3 rootwait'
setenv bootmmc          '1:1'
setenv bootcmd_sd       'ext4load mmc ${bootmmc} 0x40007fc0 boot/uImage+dtb'
```

```
setenv bootcmd 'setenv bootargs ${bootargs_console} ${bootargs_video} ${bootargs_root};
run bootcmd_sd; bootm 0x40007fc0'
saveenv

NB: replace bootmmc value '1:1' with '0:1' or '2:1' to access the good slot
    use 'ext4ls mmc XXX:1' to test access
devel@bsp-devkit:/xdt/build$ ls -lh $XDT_WORKSPACE
-rw-r--r-- 1 devel devel 2.0G Feb 15 14:13 agl-demo-platform-porter-sdcard.img
devel@bsp-devkit:/xdt/build$
```

After the disk image is created, we can copy it on the SD card itself using an SD card adapter. To do so, we need to gain access to the SD card image file from our host machine.

If you already share a directory between your host machine and the container (as described in section 3.5), this state is already reached and you go directly on sections relating the SD card image installation.

Otherwise, you need to copy the SD card image file out of the container and into your host machine using SSH protocol:

- On Linux and Mac OS X hosts, you can use the "scp" command, which is part of the OpenSSH project,
- On Windows hosts, you can use the "pscp.exe" binary, which is part of the PuTTY[6] project.

6.2. Deployment from a Linux or Mac OS X host

Now that the SD card image is ready, the last step required is to "flash" it onto the SD card itself.

First, you need an SD card adapter connected to your host machine. Depending on your adapter type and OS, the relevant block device can change. Mostly, you can expect:

- "/dev/sd*X*" block device; usually for external USB adapters on Linux hosts,
- "/dev/mmcblk*N*": when using a laptop internal adapter on Linux hosts,
- "/dev/disk*N*": on Mac OS X hosts,

6.2.1. Linux

If you do not know which block device you should use, you can check the kernel logs using the following command to figure out what is the associated block devices:

```
$ dmesg | grep mmcblk
$ dmesg | grep sd
```

6 http://www.putty.org/

```
[...snip...]
[1131831.853434] sd 6:0:0:0: [sdb] 31268864 512-byte logical blocks: (16.0 GB/14.9 GiB)
[1131831.853758] sd 6:0:0:0: [sdb] Write Protect is on
[1131831.853763] sd 6:0:0:0: [sdb] Mode Sense: 4b 00 80 08
[1131831.854152] sd 6:0:0:0: [sdb] No Caching mode page found
[1131831.854157] sd 6:0:0:0: [sdb] Assuming drive cache: write through
[1131831.855174] sd 6:0:0:0: [sdb] No Caching mode page found
[1131831.855179] sd 6:0:0:0: [sdb] Assuming drive cache: write through
[...snip...]
$
```

In this example, no "mmcblk" device where found, but a 16.0GB disk was listed and can be accessed on "**/dev/sdb**" which in our case is the physical SD card capacity.

The command 'lsblk' is also a good solution to explore block devices:

```
$ lsblk
NAME            MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda               8:0    0 931.5G  0 disk
├─sda1            8:1    0     8G  0 part /
├─sda2            8:2    0    16G  0 part [SWAP]
└─sda3            8:3    0 907.5G  0 part
  ├─vg0-usr     254:0    0    32G  0 lvm  /usr
  ├─vg0-data    254:1    0   200G  0 lvm  /data
  ├─vg0-home    254:2    0   100G  0 lvm  /home
  ├─vg0-var     254:3    0     8G  0 lvm  /var
  └─vg0-docker  254:4    0   100G  0 lvm  /docker
sdb               8:16   0 223.6G  0 disk
└─sdb1            8:17   0 223.6G  0 part /ssd
sdc               8:32   1   3.7G  0 disk
└─sdc1            8:33   1     2G  0 part
sr0              11:0    1  1024M  0 rom
```

In this example, the 4GB device "/dev/sdc" is listed as removable (column RM) and corresponds to a SD Card plugged into an USB card reader.

Finally, as we know the block device which corresponds to our SD card, we can raw-copy the image on it using the following command **from your host terminal**: (replace */dev/sdZ* by the appropriate device)

```
$ xzcat ~/mirror/agl-demo-platform-porter-20XXYYZZxxyyzz.raw.xz | sudo dd of=/dev/sdZ
bs=1M
2048+0 records in
2048+0 records out
2147483648 bytes (2.0 GB) copied, 69 s, 39.2 MB/s
$ sync
```

This will take few minutes to copy and sync. You should not remove the card from its slot until both commands succeed.

Once it is finished, you can unplug the card and insert it in the micro-SD card slot on the Porter board, and perform a power cycle to start your new image on the target.

NB: The output format is also suitable to bmaptool utility (source code available here: https://github.com/01org/bmap-tools): this significantly speeds up the copy as only relevant data are written on the Sdcard (filesystem "holes" are not written). It's also supporting direct access to URLs pointing to compressed images.

6.2.2. Mac OS X©

If you do not know which block device you should use, you can use the *diskutil* tool to list them:

```
$ diskutil list

[...snip...]
/dev/disk2
      #:                       TYPE   NAME          SIZE          IDENTIFIER
      0: Fdisk_partition_scheme                     7.9 GB        disk2
      1:                       Linux                7.9 GB        disk2s1
[...snip…]
$
```

In this example, we have a 8.0GB disk which can be accessed on "**/dev/disk2**" which in our case is the physical SD card capacity.

Finally, as we know the block device which accesses our SD card, we can raw-copy the image on it using the following command from your host terminal:

```
$ xzcat ~/mirror/agl-demo-platform-porter-20XXYYZZxxyyzz.raw.xz | sudo dd of=/dev/disk2
bs=1M
2048+0 records in
2048+0 records out
2147483648 bytes (2.0 GB) copied, 69 s, 39.2 MB/s
$ sync
```

This will take few minutes to copy and sync. You should not remove the card from its slot until both commands succeed.

Once it is finished, you can unplug the card and insert it in the micro-SD card slot on the Porter board, and perform a power cycle to start your new image on the target.

6.3. Deployment from a Windows host

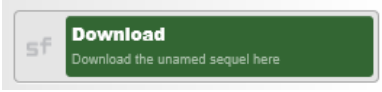
Now that the SD card image is ready, the last step required is to "flash" it onto the SD card itself.

First, you need an SD card adapter connected to your host machine.

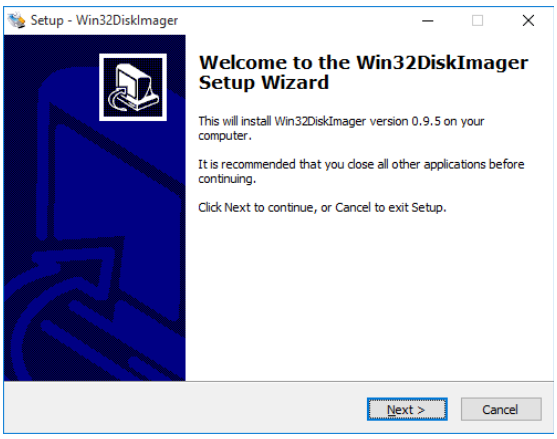
We will then use the *Win32DiskImager* program which we will download at this URL:

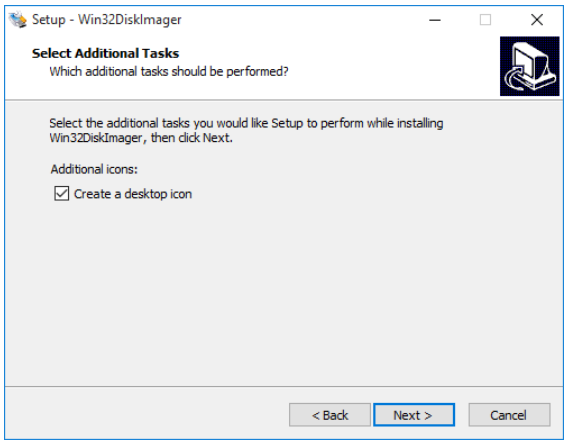
http://sourceforge.net/projects/win32diskimager/

and by clicking on this button:

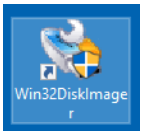


We will then install it:



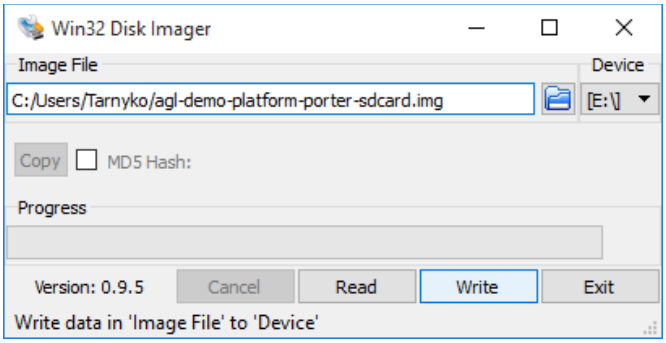


And then start it with its icon:



We can then click on the "blue folder" button to select our .img file (uncompress the XZ image first using utilities like 7zip for example).

After having verified that the drive letter on the right matches our SD card reader, we click on the "**Write**" button to start the flashing process.



This will take few minutes to copy and sync. You should not remove the card from its slot until both commands succeed.

Once it is finished, you can unplug the card and insert it in the micro-SD card slot on the Porter board, and perform a power cycle to start your new image on the target.

7. AGL SDK compilation and installation

Now that we have both a finalized development container and a deployed Porter image, let us create and install the SDK (Software Development Kit), so that we can develop new components for our image.

Going back to the container, let's generate our SDK files:

```
devel@bsp-devkit:~$ bitbake agl-demo-platform-crosssdk
```

This will take some time to complete.

Alternatively, you can download a prebuilt SDK file suitable for AGL 2.0 on IoT.bzh website:

```
devel@bsp-devkit:~$ mkdir -p /xdt/build/tmp/deploy/sdk
devel@bsp-devkit:~$ cd /xdt/build/tmp/deploy/sdk
devel@bsp-devkit:/xdt/build/tmp/deploy/sdk$ wget \
http://iot.bzh/download/public/2016/bsp/poky-agl-glibc-x86_64-agl-demo-
platform-crosssdk-cortexa15hf-vfp-neon-toolchain-1.0+snapshot.sh
```

Once you have the prompt again, let's install our SDK to its final destination. For this, run the script 'install_sdk' with the SDK auto-installable archive as argument:

```
devel@bsp-devkit:~$ install_sdk /xdt/build/tmp/deploy/sdk
/poky-agl-glibc-x86_64-agl-demo-platform-crosssdk-
cortexa15hf-vfp-neon-toolchain-1.0+snapshot.sh
```

The SDK files should be now installed in **/xdt/sdk**:

```
devel@bsp-devkit:~$ tree -L 2 /xdt/sdk
/xdt/sdk/
|-- environment-setup-cortexa15hf-vfp-neon-poky-linux-gnueabi
|-- site-config-cortexa15hf-vfp-neon-poky-linux-gnueabi
|-- sysroots
|   |-- cortexa15hf-vfp-neon-poky-linux-gnueabi
|   `-- x86_64-pokysdk-linux
`-- version-cortexa15hf-vfp-neon-poky-linux-gnueabi
```

You can now use them to develop new services, and native/HTML applications.

Please refer to the document entitled "Build Your 1st AGL Application" to learn how to do this.